Advanced MPI Programming

Tutorial at SC17, November 2017

Latest slides and code examples are available at

www.mcs.anl.gov/~thakur/sc17-mpi-tutorial

Pavan Balaji

Argonne National Laboratory

Email: balaji@anl.gov

Web: www.mcs.anl.gov/~balaji

William Gropp

University of Illinois, Urbana-Champaign

Email: wgropp@illinois.edu

Web: www.cs.illinois.edu/~wgropp

Torsten Hoefler

ETH Zurich

Email: htor@inf.ethz.ch

Web: http://htor.inf.ethz.ch/

Rajeev Thakur

Argonne National Laboratory

Email: thakur@anl.gov

Web: www.mcs.anl.gov/~thakur

About the Speakers

- **Pavan Balaji**: Computer Scientist, Argonne National Laboratory
- **William Gropp**: Director, NCSA; Professor, University of Illinois, Urbana-Champaign
- **Torsten Hoefler**: Associate Professor, ETH Zurich
- **Rajeev Thakur**: Senior Computer Scientist, Argonne National Laboratory

- All four of us are deeply involved in MPI standardization (in the MPI Forum) and in MPI implementation

Outline

Morning

- Introduction
  - MPI-1, MPI-2, MPI-3
- Running example: 2D stencil code
  - Simple point-to-point version
- Derived datatypes
  - Use in 2D stencil code
- One-sided communication
  - Basics and new features in MPI-3
  - Use in 2D stencil code
  - Advanced topics
    - Global address space communication

Afternoon

- MPI and Threads
  - Thread safety specification in MPI
  - How it enables hybrid programming
  - Hybrid (MPI + shared memory) version of 2D stencil code
  - MPI + accelerators
- Nonblocking collectives
  - Parallel FFT example
- Process topologies
  - 2D stencil example
- Neighborhood collectives
  - 2D stencil example
- Recent efforts of the MPI Forum
- Conclusions

MPI-1

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
- MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, etc
- MPI-1 was defined (1994) by a broadly based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)

MPI-2

- Same process of definition by MPI Forum
- MPI-2 is an extension of MPI
  - Extends the message-passing model
    - Parallel I/O
    - Remote memory operations (one-sided)
    - Dynamic process management
  - Adds other functionality
    - C++ and Fortran 90 bindings
      - similar to original C and Fortran-77 bindings
    - External interfaces
    - Language interoperability
    - MPI interaction with threads

Timeline of the MPI Standard

- MPI-1 (1994), presented at SC'93
  - Basic point-to-point communication, collectives, datatypes, etc
- MPI-2 (1997)
  - Added parallel I/O, Remote Memory Access (one-sided operations), dynamic processes, thread support, C++ bindings, …
- ---- Stable for 10 years ----
- MPI-2.1 (2008)
  - Minor clarifications and bug fixes to MPI-2
- MPI-2.2 (2009)
  - Small updates and additions to MPI 2.1
- MPI-3.0 (2012)
  - Major new features and additions to MPI
- MPI-3.1 (2015)
  - Minor updates and fixes to MPI 3.0

Overview of New Features in MPI-3

- Major new features
  - Nonblocking collectives
  - Neighborhood collectives
  - Improved one-sided communication interface
  - Tools interface
  - Fortran 2008 bindings
- Other new features
  - Matching Probe and Recv for thread-safe probe and receive
  - Noncollective communicator creation function
  - “const” correct C bindings
  - Comm_split_type function
  - Nonblocking Comm_dup
  - Type_create_hindexed_block function
- C++ bindings removed
- Previously deprecated functions removed
- MPI 3.1 added nonblocking collective I/O functions

Status of MPI-3.1 Implementations

	MPICH	MVAPICH	Open MPI	Cray	Tianhe	Intel		IBM			HPE	Fujitsu	MS	MPC	NEC	Sunway	RIKEN	AMPI
						IMPI	MPICH-OFI	BG/Q (legacy) [1]	PE (legacy) [2]	Spectrum								
NBC	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Nbr. Coll.	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔
RMA	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	(*)	✔	✔	✔	✔	Q2 '18
Shared mem	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	Q1 '18
MPI_T	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*	✔	✔	✔	✔	Q2 '18
Comm-create group	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*	✔	✔	✔	✔	✔
F08 Bindings	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	✘	✔	✔	✔	✔	Q2 '18
New Dtypes	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Large Counts	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
MProbe	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	Q1 '18
NBC I/O	✔	✔	✔	✔	✘	✔	✔	✘	✘	✔	✔	✘	✘	*	✔	✘	✔	Q3 '18

Release dates are estimates; subject to change at any time "✘" indicates no publicly announced plan to implement/support that feature

Platform-specific restrictions might apply to the supported features

[1] Open Source but unsupported [2] No MPI_T variables exposed * Under development (*) Partly done

Important considerations while using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

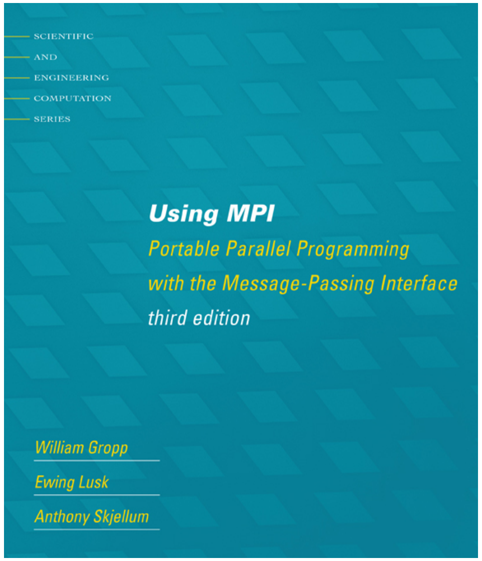
Web Pointers

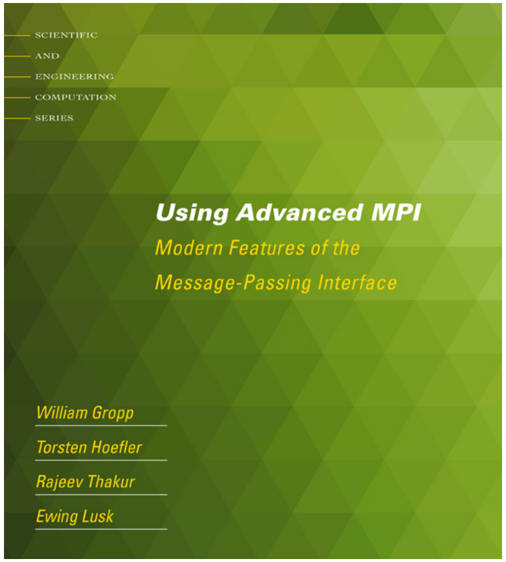
- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: https://msdn.microsoft.com/en-us/library/bb524831%28v=vs.85%29.aspx
  - Open MPI : http://www.open-mpi.org/
  - IBM MPI, Cray MPI, HP MPI, TH MPI, …
- Several MPI tutorials can be found on the web

New Tutorial Books on MPI

- For basic MPI
  - ***Using MPI, 3rd edition, 2014***, by William Gropp, Ewing Lusk, and Anthony Skjellum
  - https://mitpress.mit.edu/using-MPI-3ed
- For advanced MPI, including MPI-3
  - ***Using Advanced MPI, 2014,*** by William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk
  - https://mitpress.mit.edu/using-advanced-MPI

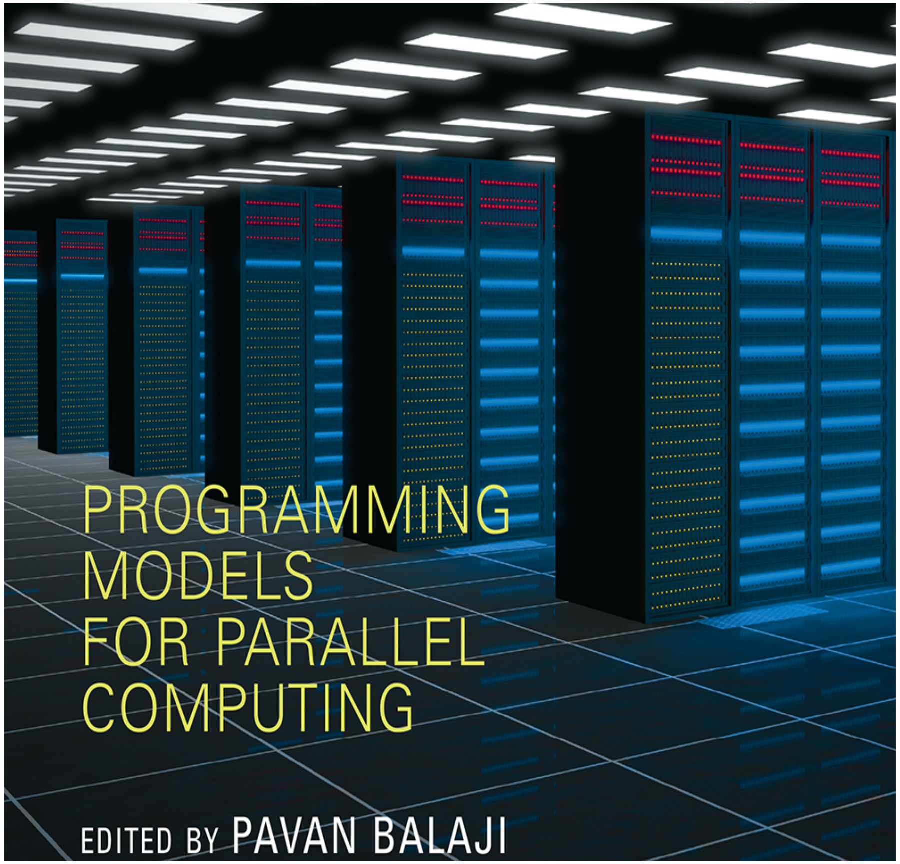




New Book on Parallel Programming Models

Edited by Pavan Balaji

- ***MPI:*** W. Gropp and R. Thakur
- ***GASNet:*** P. Hargrove
- ***OpenSHMEM:*** J. Kuehn and S. Poole
- ***UPC:*** K. Yelick and Y. Zheng
- ***Global Arrays:*** S. Krishnamoorthy, J. Daily, A. Vishnu, and B. Palmer
- ***Chapel:*** B. Chamberlain
- ***Charm++:*** L. Kale, N. Jain, and J. Lifflander
- ***ADLB:*** E. Lusk, R. Butler, and S. Pieper
- ***Scioto:*** J. Dinan
- ***SWIFT:*** T. Armstrong, J. M. Wozniak, M. Wilde, and I. Foster
- ***CnC:*** K. Knobe, M. Burke, and F. Schlimbach
- ***OpenMP:*** B. Chapman, D. Eachempati, and S. Chandrasekaran
- ***Cilk Plus:*** A. Robison and C. Leiserson
- ***Intel TBB:*** A. Kukanov
- ***CUDA:*** W. Hwu and D. Kirk
- ***OpenCL:*** T. Mattson



https://mitpress.mit.edu/models

Our Approach in this Tutorial

- Example driven
  - 2D stencil code used as a running example throughout the tutorial
  - Other examples used to illustrate specific features
- We will walk through actual code
- We assume familiarity with basic concepts of MPI-1

Regular Mesh Algorithms

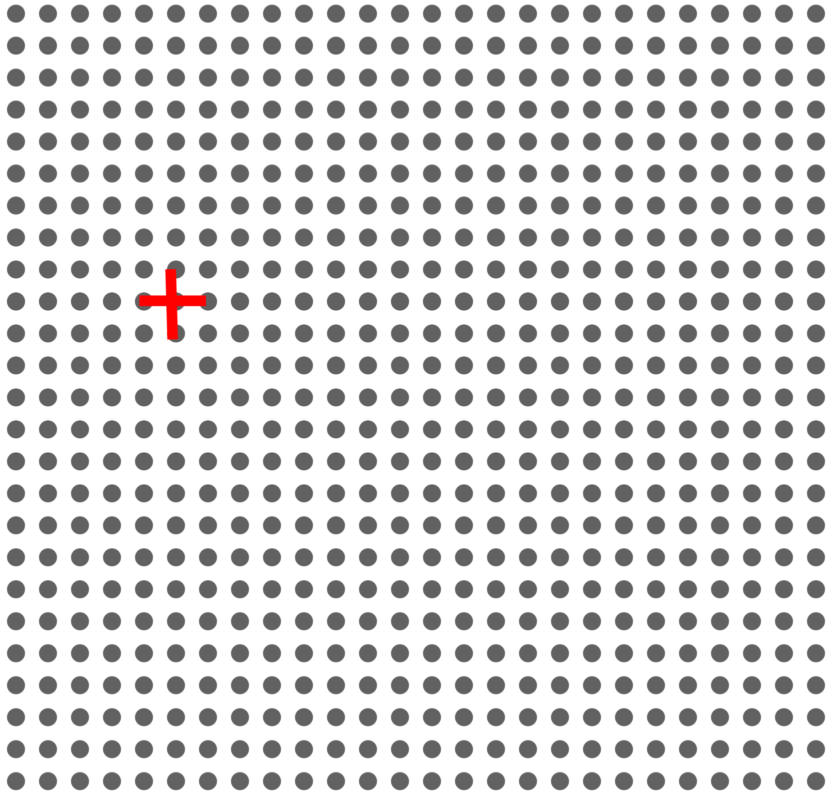
- Many scientific applications involve the solution of partial differential equations (PDEs)
- Many algorithms for approximating the solution of PDEs rely on forming a set of difference equations
  - Finite difference, finite elements, finite volume
- The exact form of the difference equations depends on the particular method
  - From the point of view of parallel programming for these algorithms, the operations are the same

Poisson Problem

- To approximate the solution of the Poisson Problem $\nabla^2 u = f$ on the unit square, with u defined on the boundaries of the domain (Dirichlet boundary conditions), this simple 2nd order difference scheme is often used:
  - $(U(x+h,y) - 2U(x,y) + U(x-h,y)) / h^2 +$
    $(U(x,y+h) - 2U(x,y) + U(x,y-h)) / h^2 = f(x,y)$
    - Where the solution U is approximated on a discrete grid of points x=0, h, 2h, 3h, … , (1/h)h=1, y=0, h, 2h, 3h, … 1.
    - To simplify the notation, U(ih,jh) is denoted U_{ij}
- This is defined on a discrete mesh of points (x,y) = (ih,jh), for a mesh spacing “h”

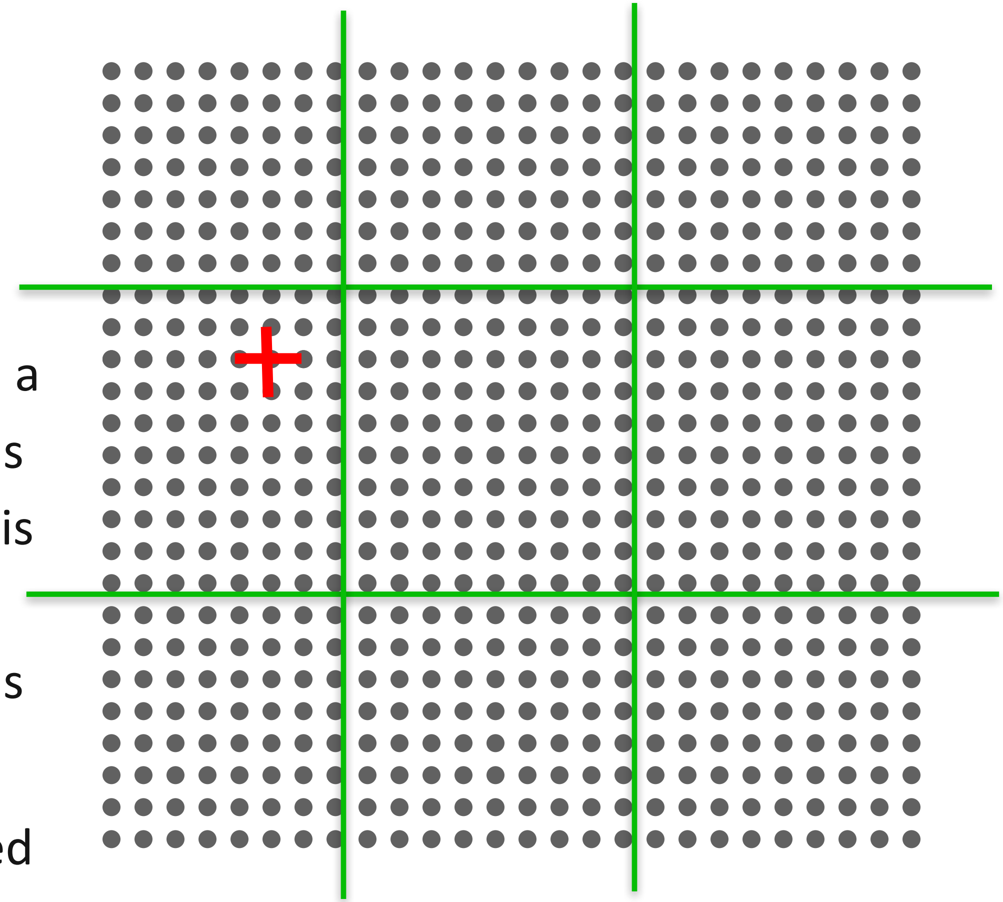
The Global Data Structure

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.

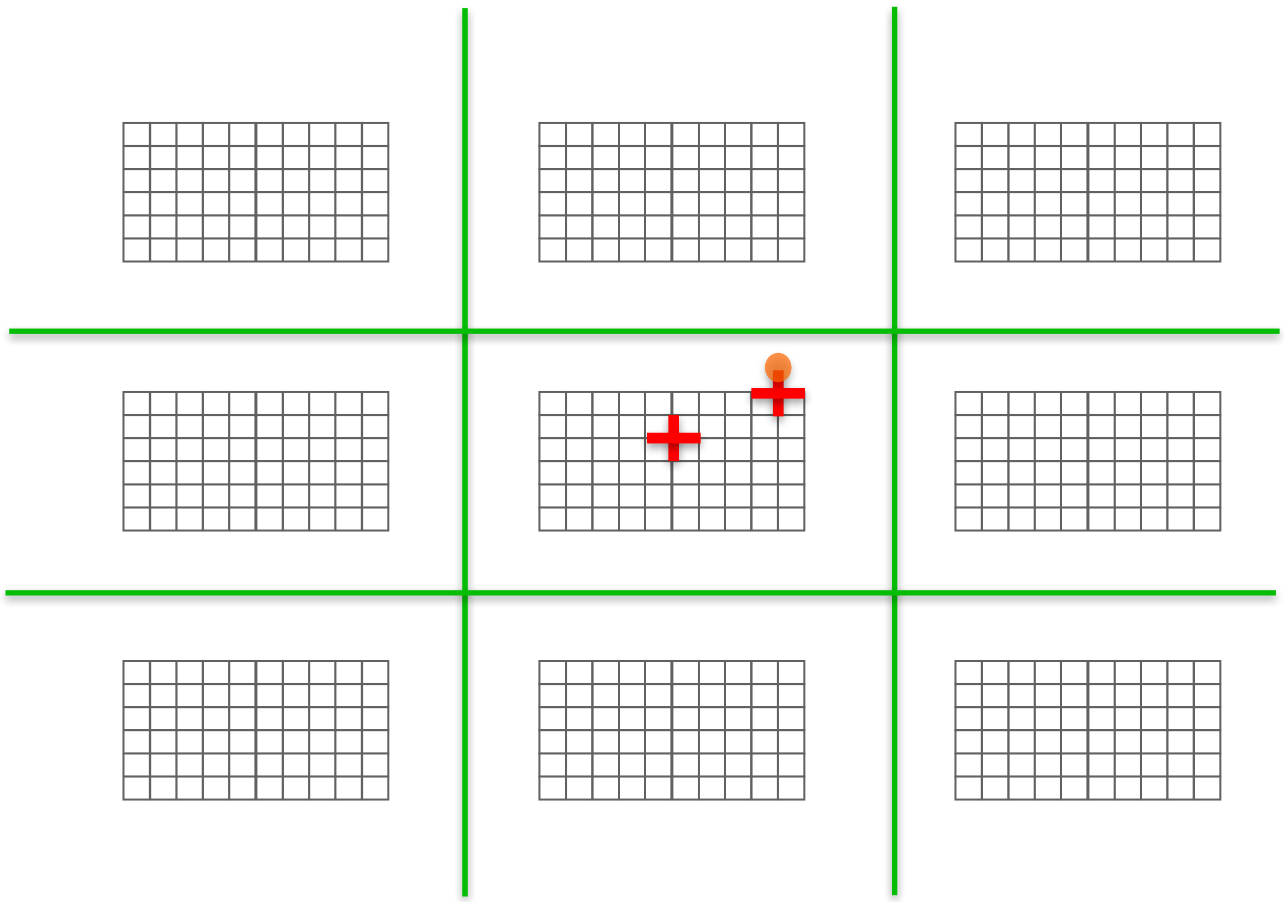


The Global Data Structure

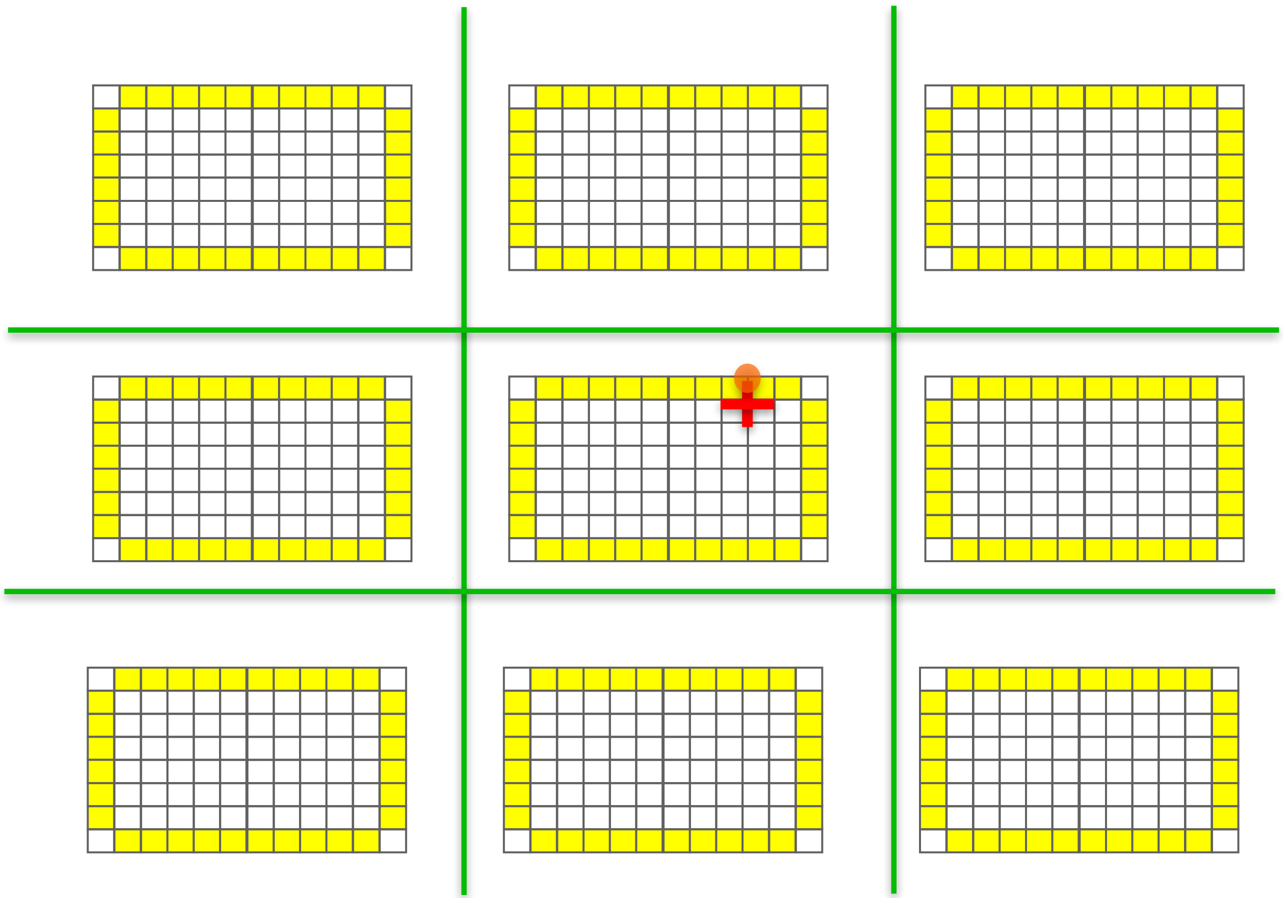
- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.
- Decompose mesh into equal sized (work) pieces



Necessary Data Transfers

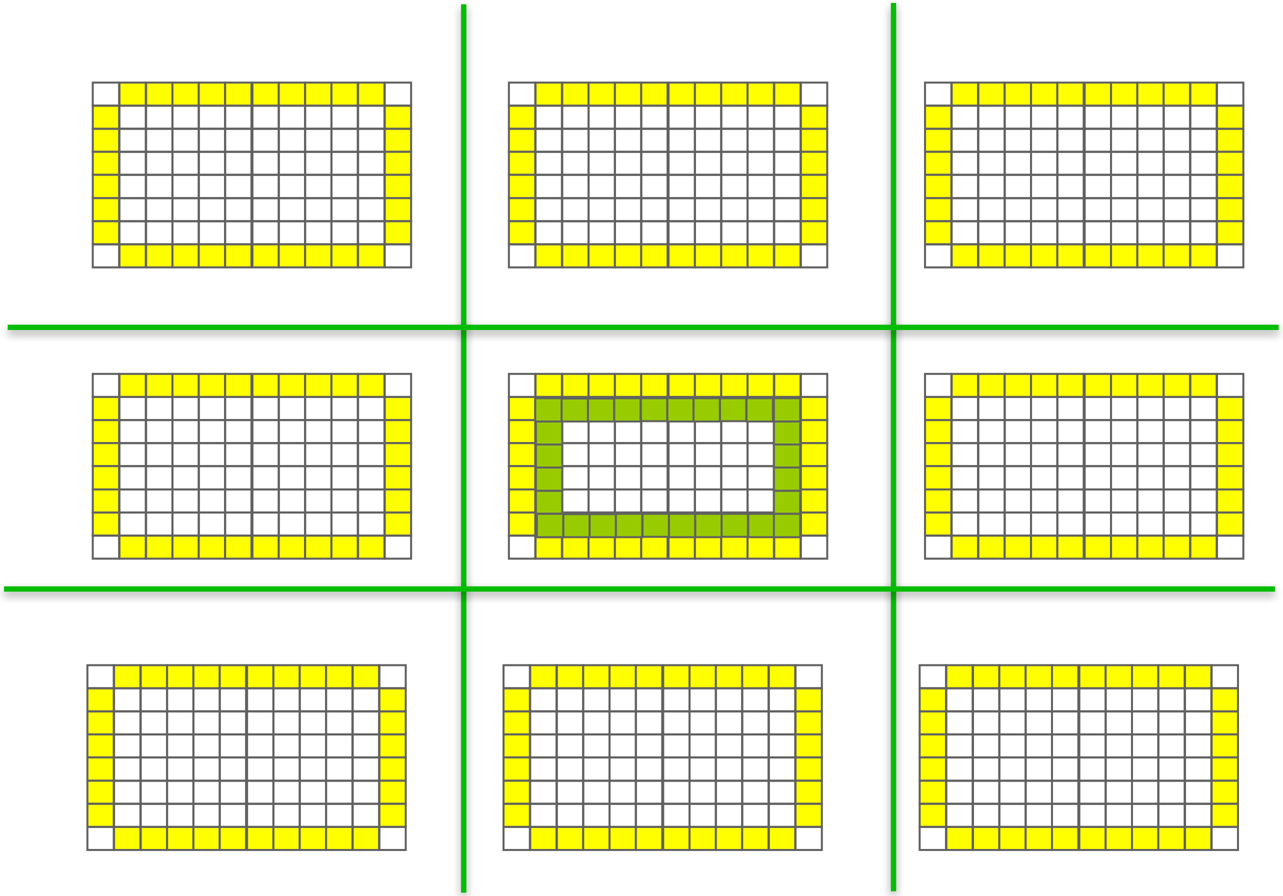


Necessary Data Transfers



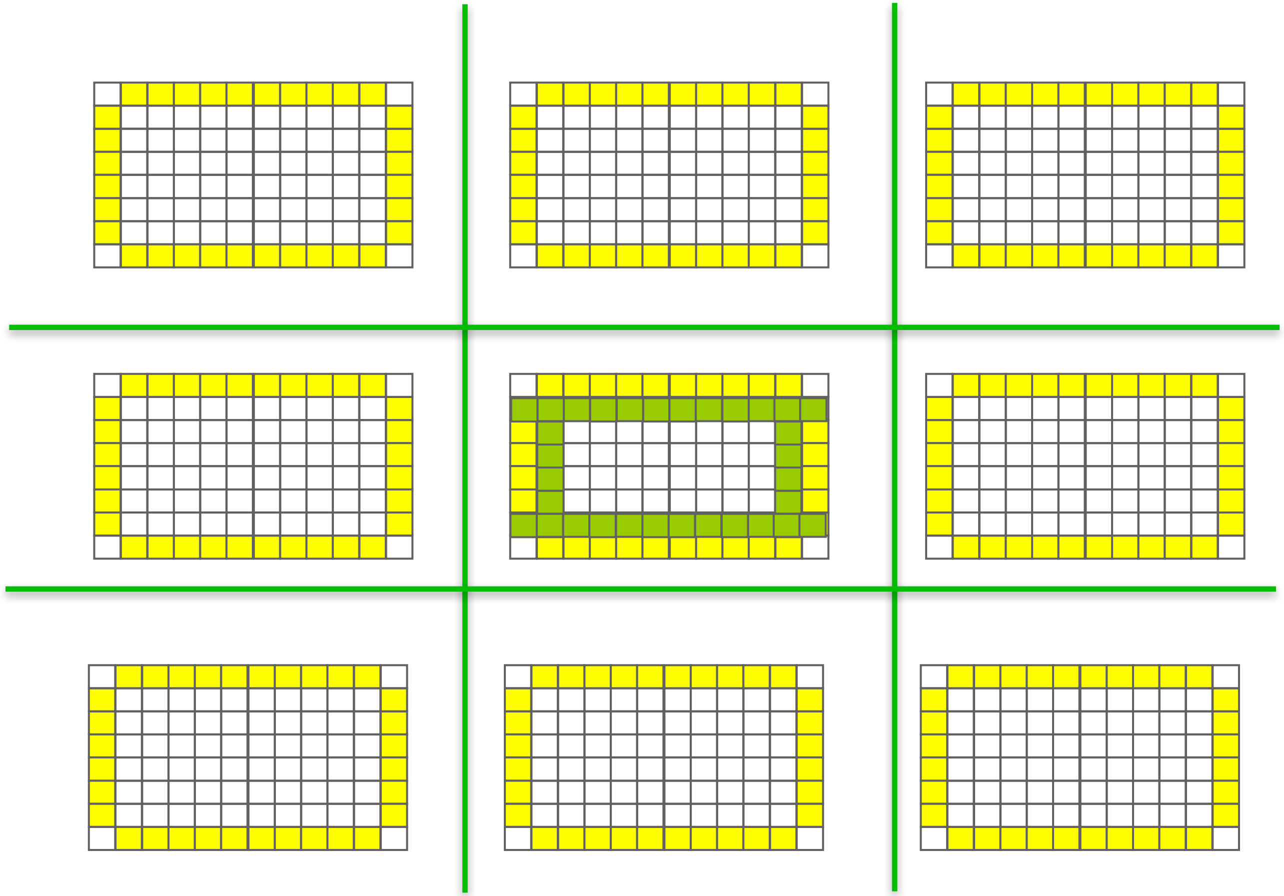
Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (5 point stencil)



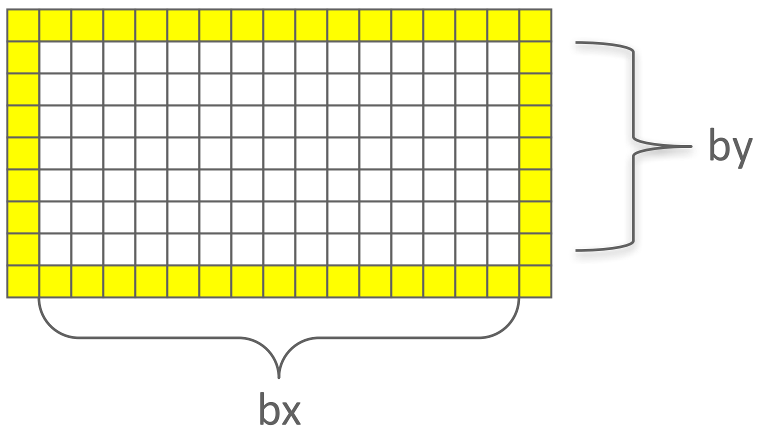
Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (9 point with trick)



The Local Data Structure

- Each process has its local “patch” of the global array
  - “bx” and “by” are the sizes of the local array
  - Always allocate a halo around the patch
  - Array allocated of size (bx+2)x(by+2)



2D Stencil Code Walkthrough

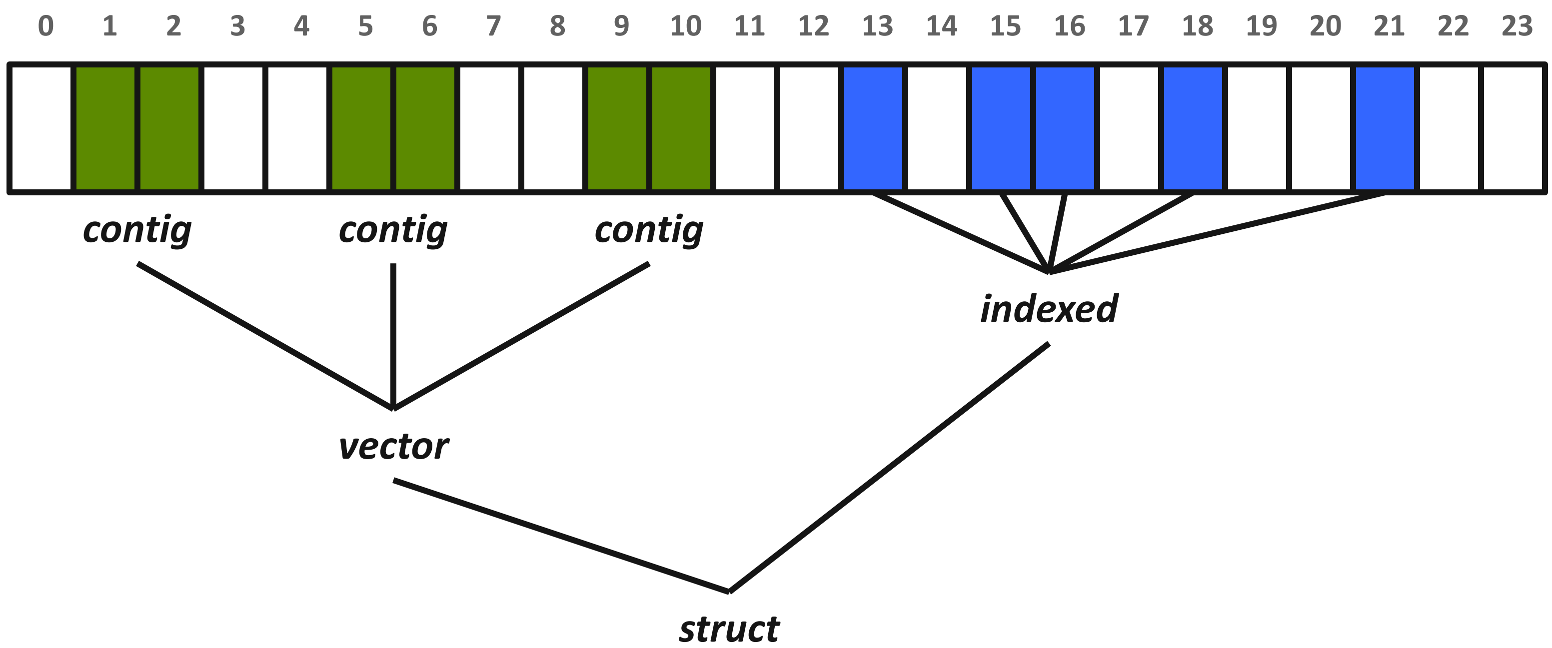
- Code can be downloaded from
  www.mcs.anl.gov/~thakur/sc17-mpi-tutorial

Datatypes

Introduction to Datatypes in MPI

- Datatypes allow users to serialize **arbitrary** data layouts into a message stream
  - Networks provide serial channels
  - Same for block devices and I/O
- Several constructors allow arbitrary layouts
  - Recursive specification possible
  - *Declarative* specification of data-layout
    - “what” and not “how”, leaves optimization to implementation (*many unexplored* possibilities!)
  - Choosing the right constructors is not always simple

Derived Datatype Example



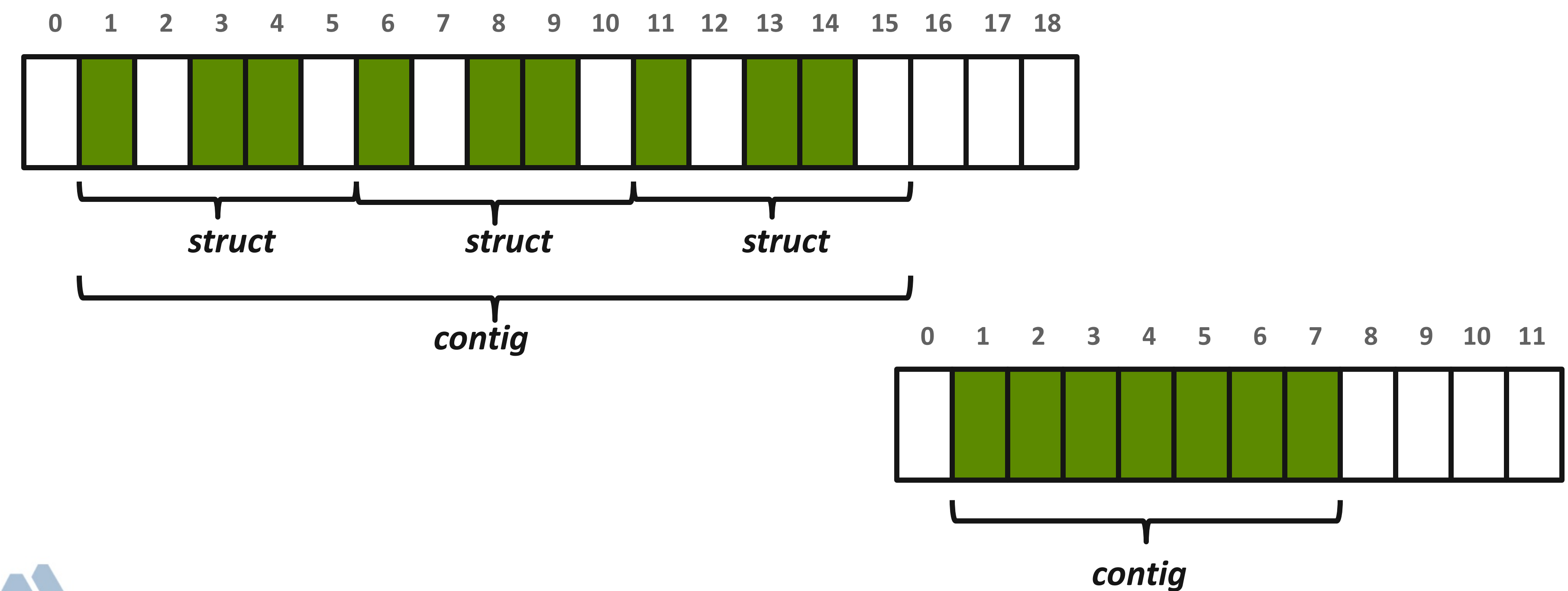
MPI's Intrinsic Datatypes

- Why intrinsic types?
  - Heterogeneity, nice to send a Boolean from C to Fortran
  - Conversion rules are complex, not discussed here
  - Length matches to language types
    - No sizeof(int) mess
- Users should generally use intrinsic types as basic types for communication and type construction
- MPI-2.2 added some missing C types
  - E.g., unsigned long long

MPI_Type_contiguous

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

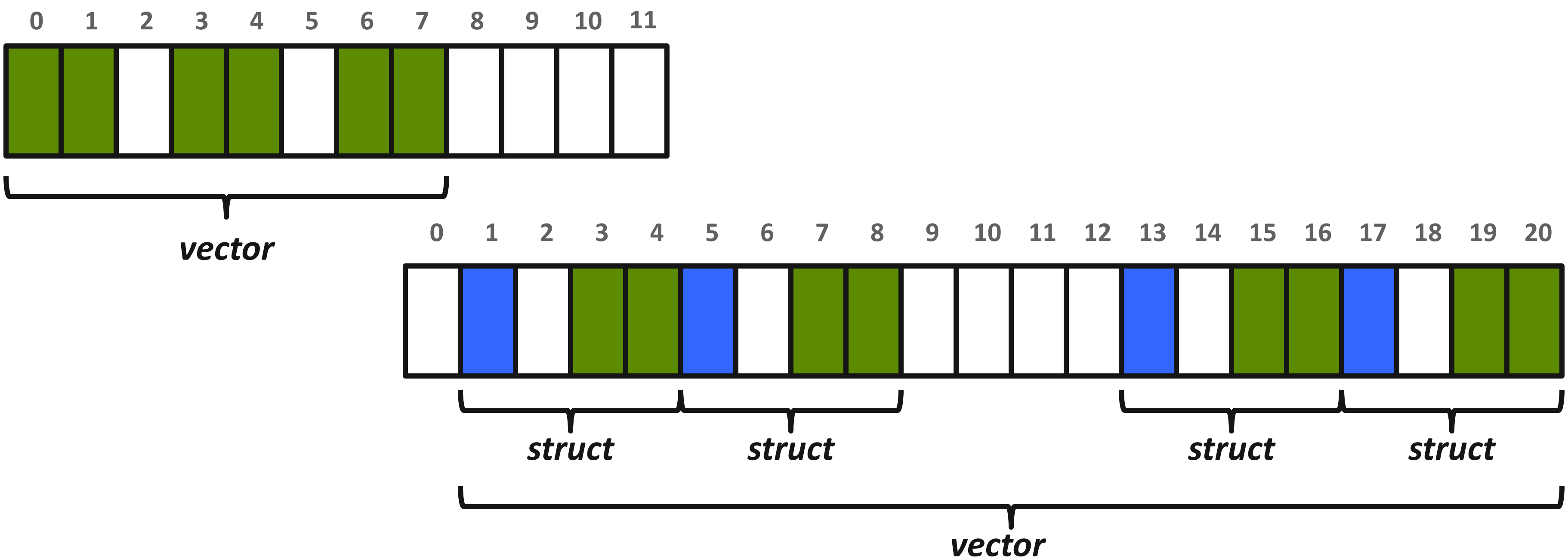
- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)



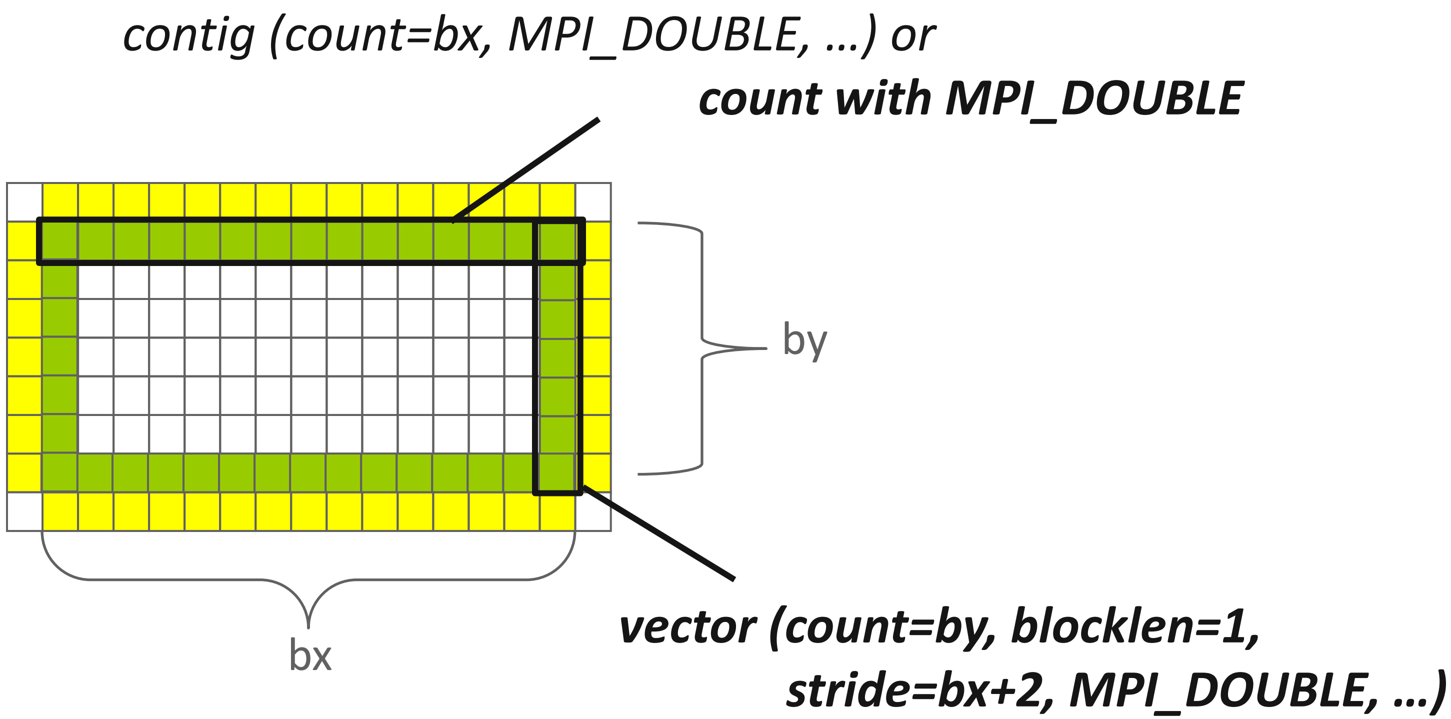
MPI_Type_vector

```
MPI_Type_vector(int count, int blocklen, int stride,
        MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Specify strided blocks of data of oldtype
- Very useful for Cartesian arrays



Use Datatype in Halo Exchange



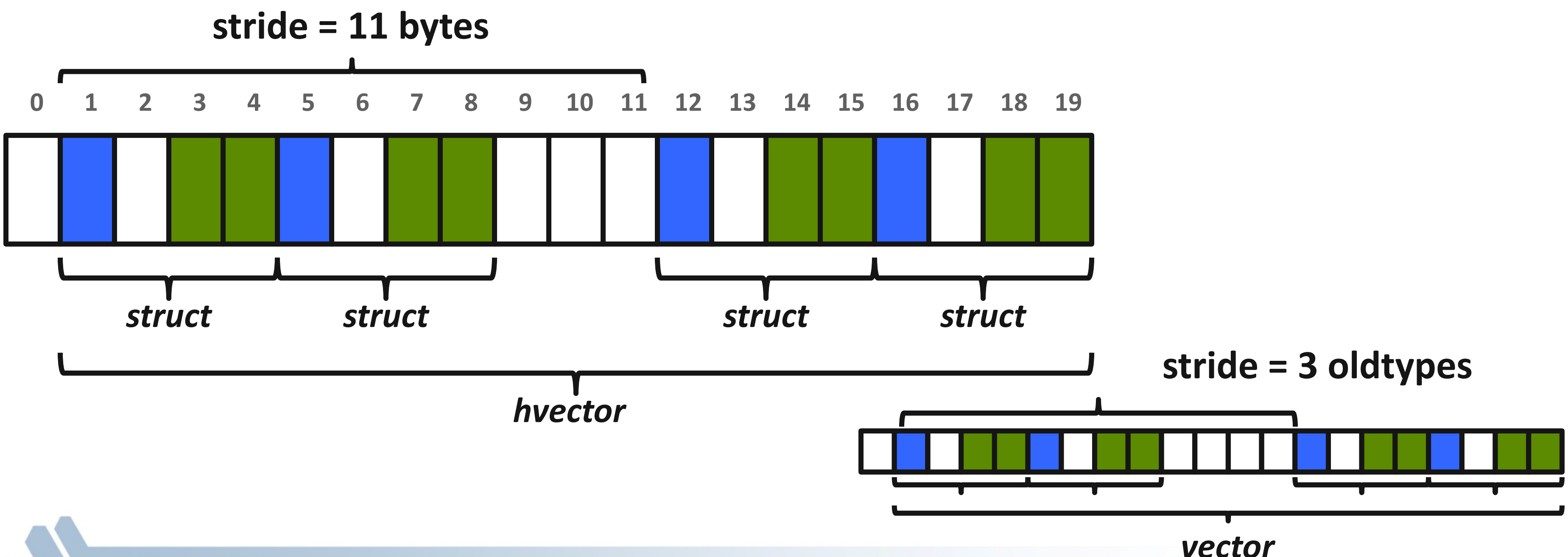
2D Stencil Code with Datatypes Walkthrough

- Code can be downloaded from
  www.mcs.anl.gov/~thakur/sc17-mpi-tutorial

MPI_Type_create_hvector

```
MPI_Type_create_hvector(int count, int blocklen, MPI_Aint stride,
        MPI_Datatype oldtype, MPI_Datatype *newtype)
```

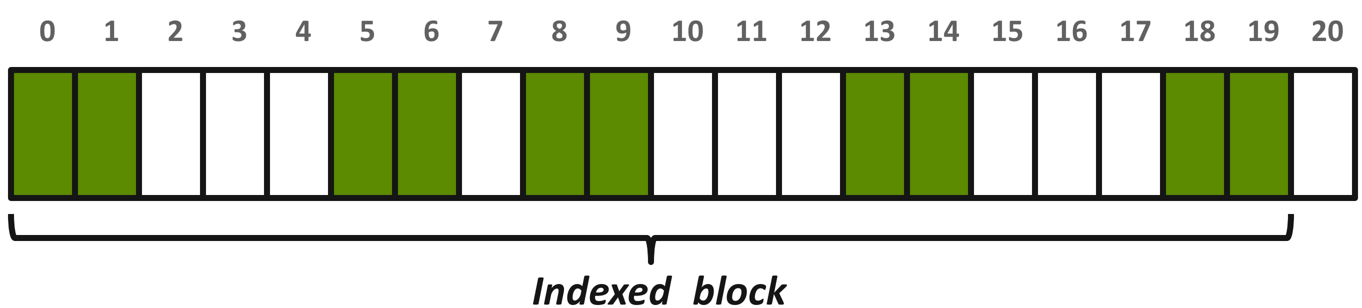
- Stride is specified in bytes instead of size of oldtype
- Useful for composition, e.g., vector of structs



MPI_Type_create_indexed_block

```
MPI_Type_create_indexed_block(int count, int blocklen,
        int *array_of_displacements,
        MPI_Datatype oldtype, MPI_Datatype *newtype)
```

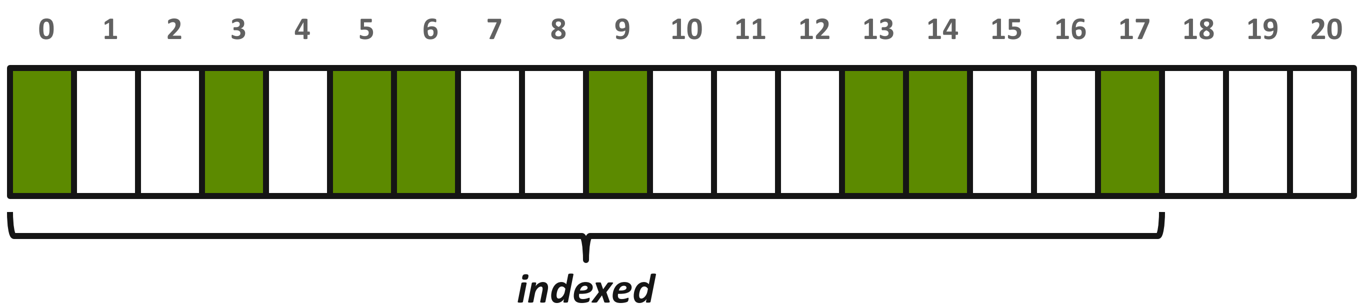
- Pulling irregular subsets of data from a single array
  - dynamic codes with index lists, expensive though!
  - blen=2
  - displs={0,5,8,13,18}



MPI_Type_indexed

```
MPI_Type_indexed(int count, int* array_of_blocklens,
         int *array_of_displacements,
         MPI_Datatype oldtype, MPI_Datatype *newtype)
```

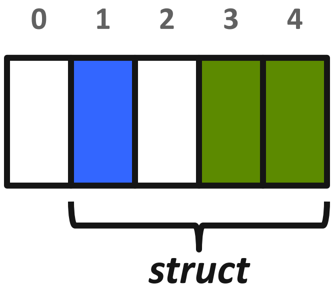
- Like indexed_block, but can have different block lengths
  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}



MPI_Type_create_struct

```
MPI_Type_create_struct(int count,
        int *array_of_blocklens,
        MPI_Aint *array_of_displacements,
        MPI_Datatype *array_of_types,
        MPI_Datatype *newtype)
```

- Most general constructor, allows different types and arbitrary arrays (also most costly)



MPI_Type_create_subarray

```
MPI_Type_create_subarray(int ndims, int* array_of_sizes,
        int *array_of_subsizes, int *array_of_starts,
        int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Convenience function for creating datatypes for array segments
- Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

MPI_Type_create_darray

```
MPI_Type_create_darray(int size, int rank, int ndims,
int array_of_gsizes[], int array_of_distribs[], int
array_of_dargs[], int array_of_psizes[], int order,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Create distributed array, supports block, cyclic and no distribution for each dimension
  - Very useful for I/O

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

MPI_BOTTOM and MPI_Get_address

- MPI_BOTTOM is the absolute zero address
  - Portability (e.g., may be non-zero in globally shared memory)
- MPI_Get_address
  - Returns address relative to MPI_BOTTOM
  - Portability (do not use “&” operator in C!)
- Very important to
  - build struct datatypes
  - If data spans multiple arrays

```
int a = 4;
float b = 9.6;
MPI_Datatype struct;

MPI_Get_address(&a, &disps[0]);
MPI_Get_address(&b, &disps[1]);

MPI_Type_create_struct(count,
                blocklens[], disps,
                oldtypes[], &struct);
```

Commit, Free, and Dup

- Types must be committed before use
  - Only the ones that are used!
  - MPI_Type_commit may perform heavy optimizations (and will hopefully)
- MPI_Type_free
  - Free MPI resources of datatypes
  - Does not affect types built from it
- MPI_Type_dup
  - Duplicates a type
  - Library abstraction (composability)

Other Datatype Functions

- Pack/Unpack
  - Mainly for compatibility to legacy libraries
  - Avoid using it yourself
- Get_envelope/contents
  - Only for expert library developers
  - Libraries such as MPITypes[1] make this easier
- MPI_Type_create_resized
  - Change extent and size (dangerous but useful)

[1] http://www.mcs.anl.gov/mpitypes/

Datatype Selection Order

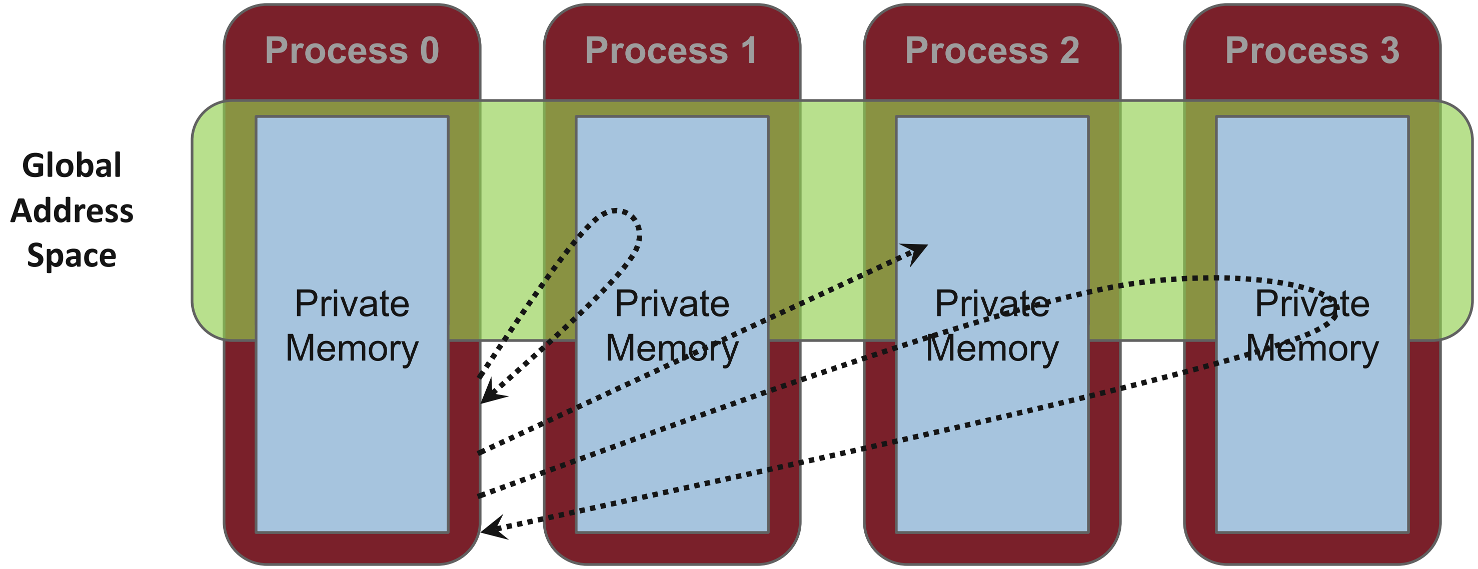
- Simple and effective performance model:
  - More parameters == slower
- **predefined < contig < vector < index_block < index < struct**
- Some (most) MPIs are inconsistent
  - But this rule is portable
- Advice to users:
  - Construct datatypes hierarchically bottom-up

W. Gropp et al.: Performance Expectations and Guidelines for MPI Derived Datatypes

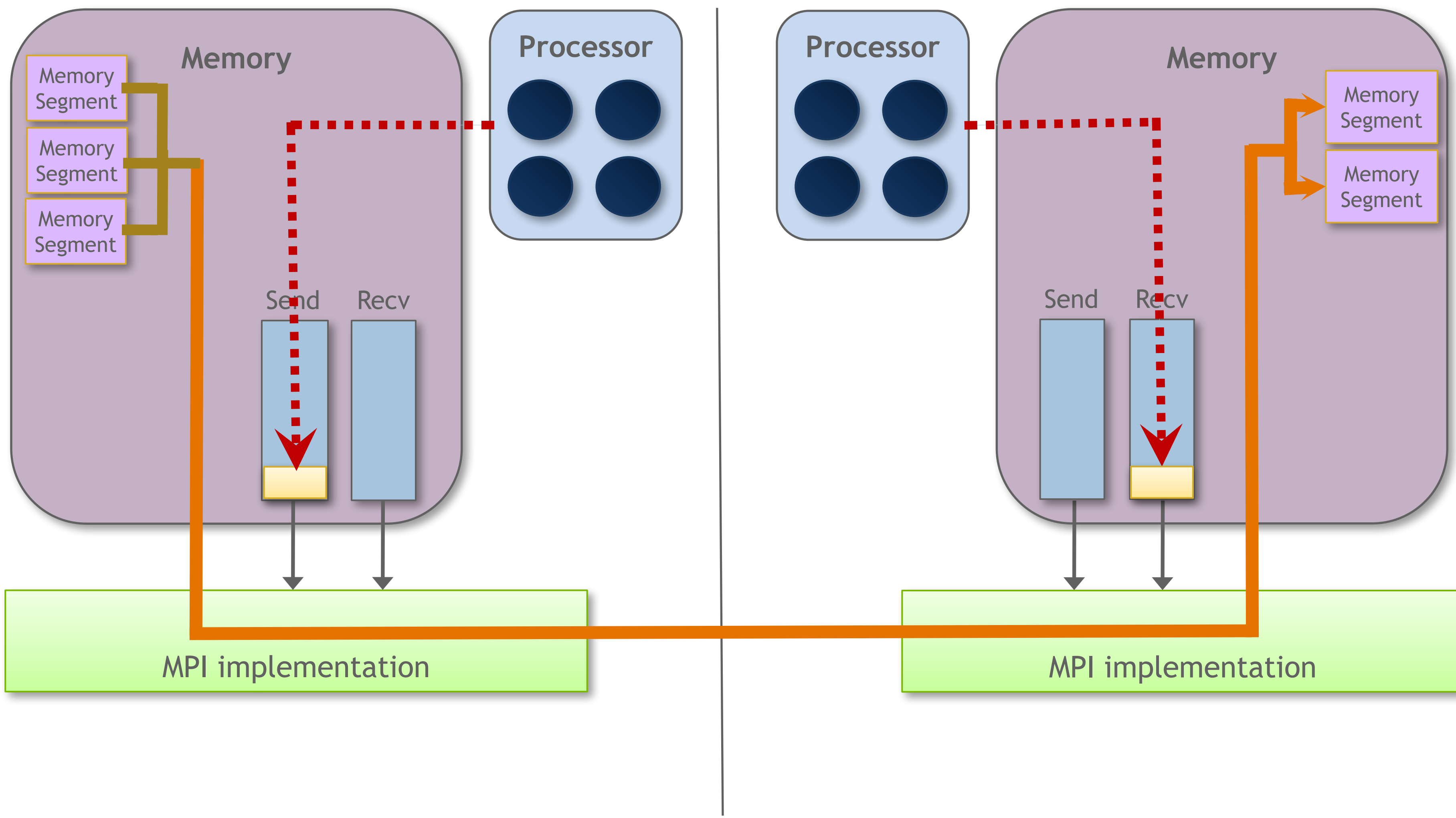
Advanced Topics: One-sided Communication

One-sided Communication

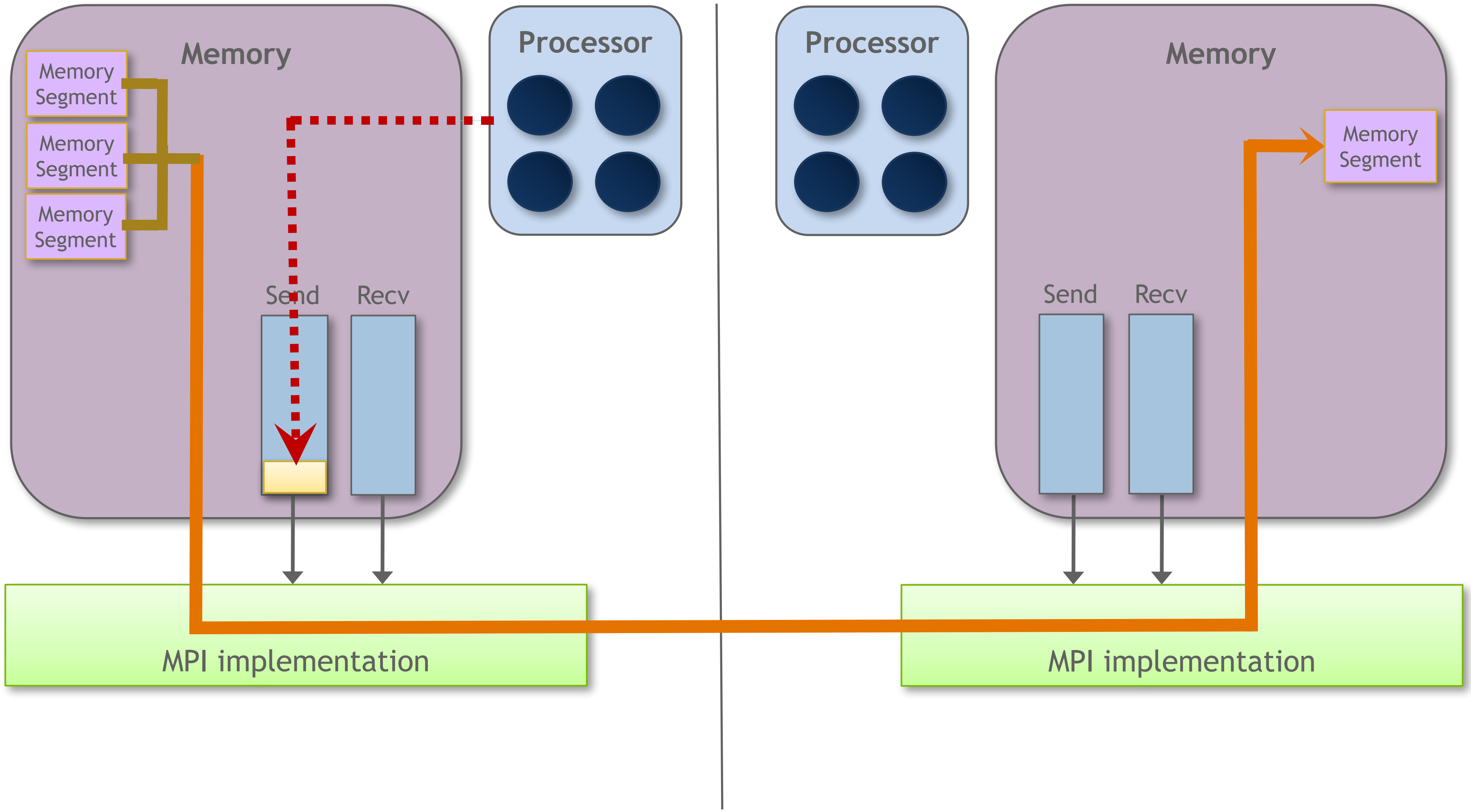
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory



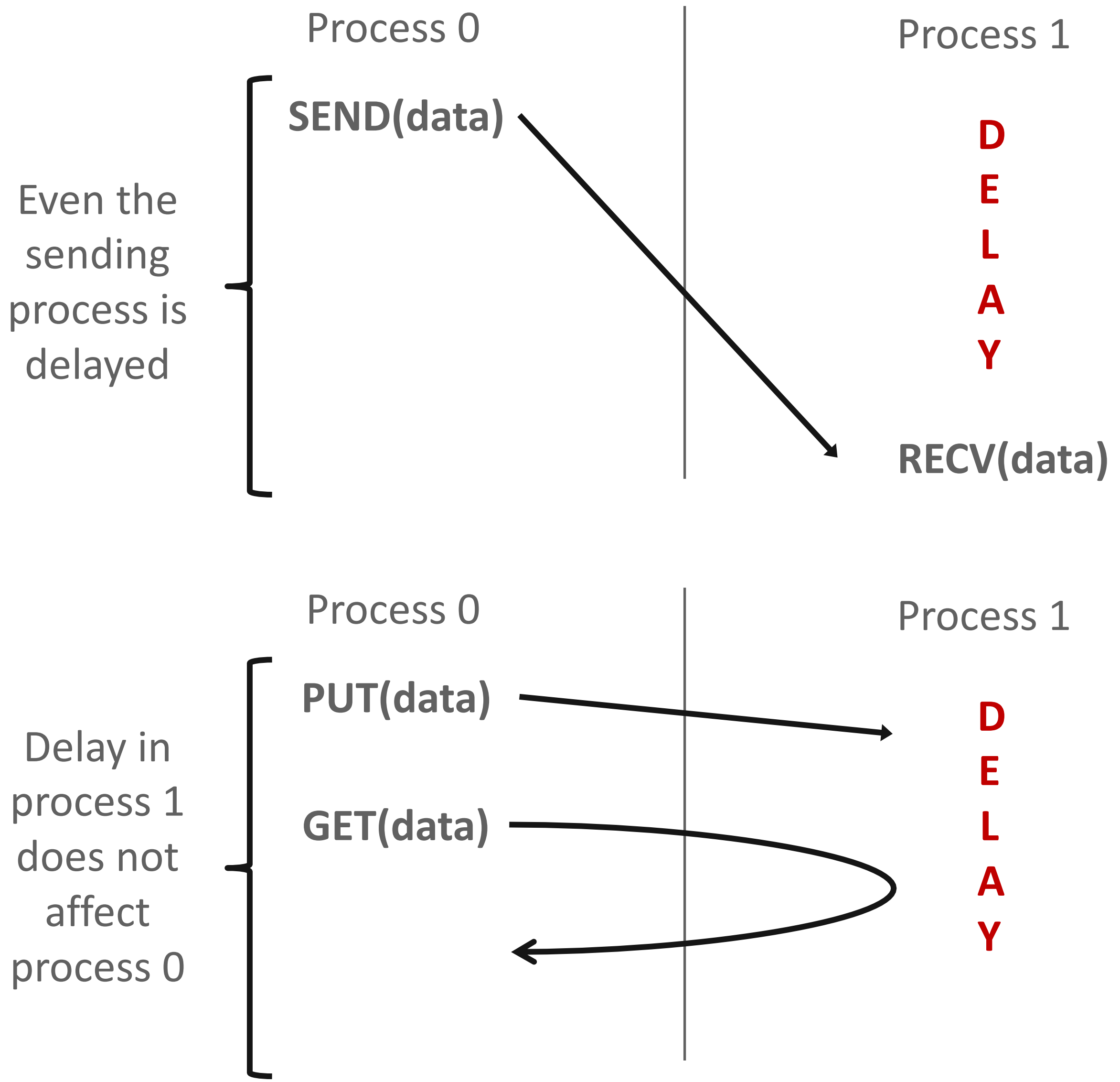
Two-sided Communication Example



One-sided Communication Example

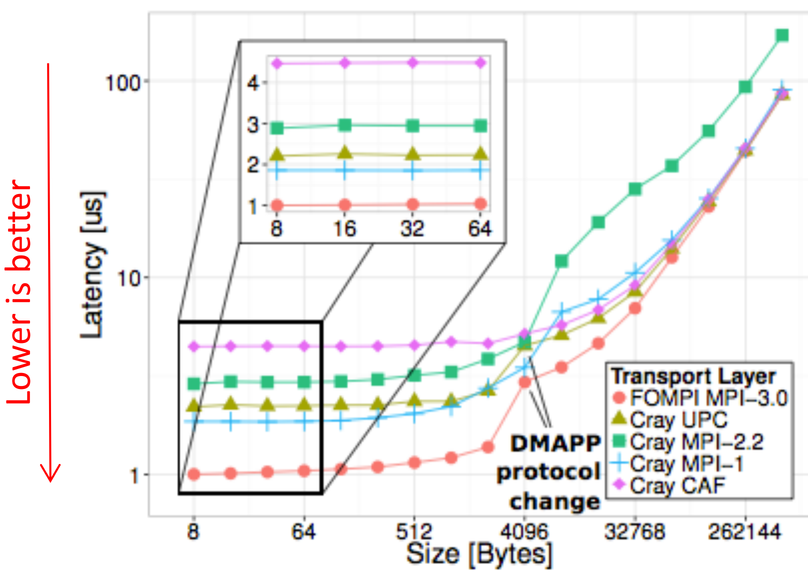


Comparing One-sided and Two-sided Programming

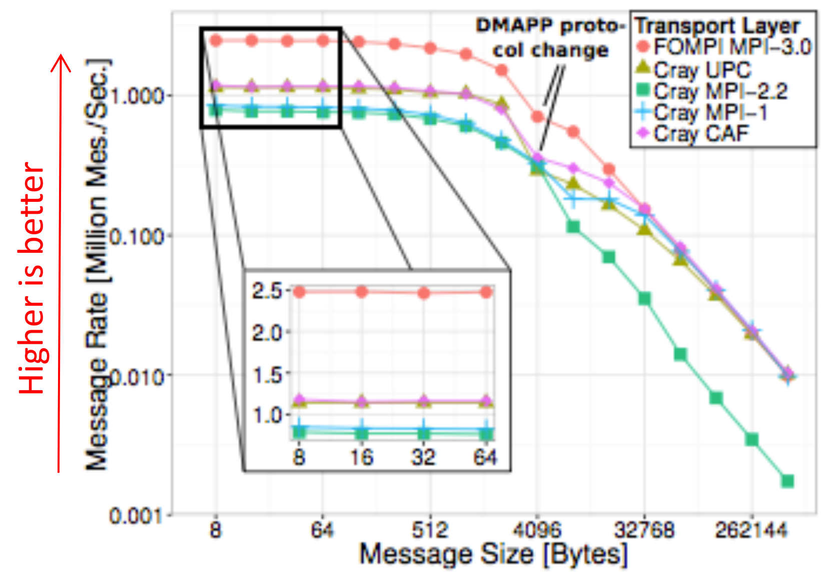


MPI RMA can be efficiently implemented

- “Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided” by Robert Gerstenberger, Maciej Besta, Torsten Hoefler (SC13 Best Paper Award)
- They implemented complete MPI-3 RMA for Cray Gemini (XK5, XE6) and Aries (XC30) systems on top of lowest-level Cray APIs
- Achieved better latency, bandwidth, message rate, and application performance than Cray’s MPI RMA, UPC, and Coarray Fortran



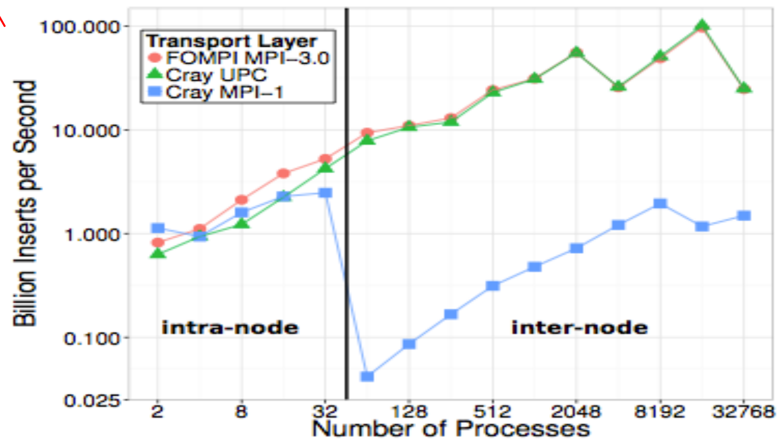
(a) Latency inter-node Put



(b) Message Rate inter-node

Application Performance with Tuned MPI-3 RMA

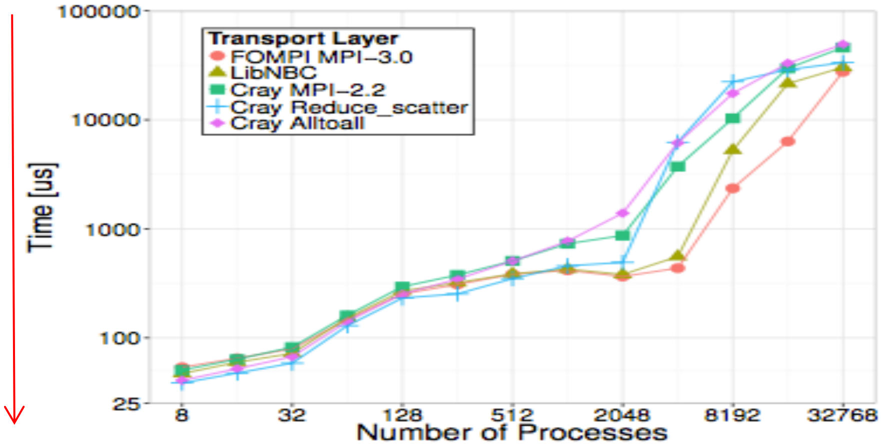
Higher is better



(a) Inserts per second for inserting 16k elements per process including synchronization.

Distributed Hash Table

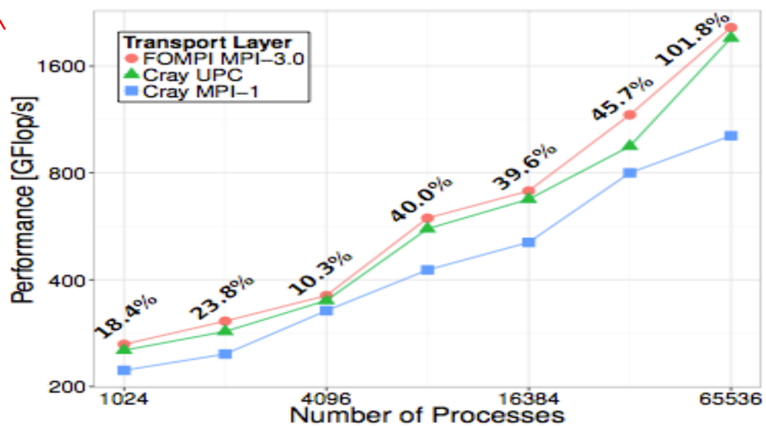
Lower is better



(b) Time to perform one dynamic sparse data exchange (DSDE) with 6 random neighbors

Dynamic Sparse Data Exchange

Higher is better



(c) 3D FFT Performance. The annotations represent the improvement of FOMPI over MPI-1.

Lower is better

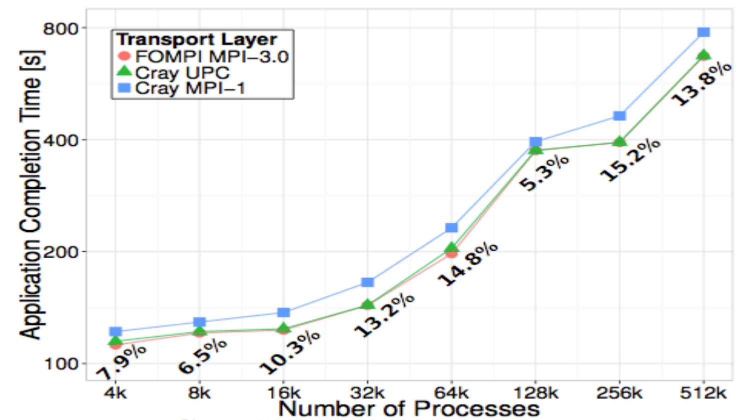


Figure 8: MILC: Full application execution time. The annotations represent the improvement of FOMPI and UPC over MPI-1.

3D FFT

MILC

Gerstenberger, Besta, Hoefler (SC13)

MPI RMA is Carefully and Precisely Specified

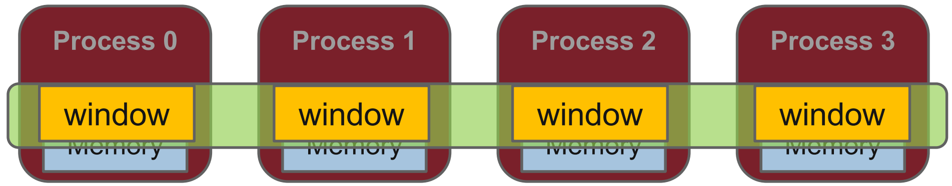
- To work on both cache-coherent and non-cache-coherent systems
  - Even though there aren't many non-cache-coherent systems, it is designed with the future in mind
- There even exists a *formal model* for MPI-3 RMA that can be used by tools and compilers for optimization, verification, etc.
  - See "Remote Memory Access Programming in MPI-3" by Hoefler, Dinan, Thakur, Barrett, Balaji, Gropp, Underwood. ACM TOPC, July 2015.
  - http://htor.inf.ethz.ch/publications/index.php?pub=201

What we need to know in MPI RMA

- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Model

Creating Public Memory

- Any memory used by a process is, by default, only locally accessible
  - X = malloc(100);



- Once the memory is allocated, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a “**window**”
  - A group of processes collectively create a “window”
- Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process

Window creation models

- Four models exist
  - MPI_WIN_ALLOCATE
    - You want to create a buffer and directly make it remotely accessible
  - MPI_WIN_CREATE
    - You already have an allocated buffer that you would like to make remotely accessible
  - MPI_WIN_CREATE_DYNAMIC
    - You don’t have a buffer yet, but will have one in the future
    - You may want to dynamically add/remove buffers to/from the window
  - MPI_WIN_ALLOCATE_SHARED
    - You want multiple processes on the same node share a buffer

MPI_WIN_ALLOCATE

```
MPI_Win_allocate(MPI_Aint size, int disp_unit,
               MPI_Info info, MPI_Comm comm, void *baseptr,
               MPI_Win *win)
```

- Create a remotely accessible memory region in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
  - baseptr - pointer to exposed local data
  - win - window (handle)

Example with MPI_WIN_ALLOCATE

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* collectively create remote accessible memory in a window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                     MPI_COMM_WORLD, &a, &win);

    /* Array 'a' is now accessible from all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

MPI_WIN_CREATE

```
MPI_Win_create(void *base, MPI_Aint size,
               int disp_unit, MPI_Info info,
               MPI_Comm comm, MPI_Win *win)
```

- Expose a region of memory in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - base - pointer to local data to expose
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
  - win - window (handle)

Example with MPI_WIN_CREATE

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int),
                   MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* Array ‘a’ is now accessibly by all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);
    MPI_Free_mem(a);
    MPI_Finalize(); return 0;
}
```

MPI_WIN_CREATE_DYNAMIC

```
MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,
                       MPI_Win *win)
```

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops
- Initially "empty"
  - Application can dynamically attach/detach memory to this window by calling MPI_Win_attach/detach
  - Application can access data on this window only after a memory region has been attached
- Window origin is MPI_BOTTOM
  - Displacements are segment addresses relative to MPI_BOTTOM
  - Must tell others the displacement after calling attach

Example with MPI_WIN_CREATE_DYNAMIC

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);
    MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* create private memory */
    a = (int *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000*sizeof(int));

    /* Array 'a' is now accessible from all processes */

    /* undeclare remotely accessible memory */
    MPI_Win_detach(win, a);  free(a);
    MPI_Win_free(&win);

   MPI_Finalize(); return 0;
}
```

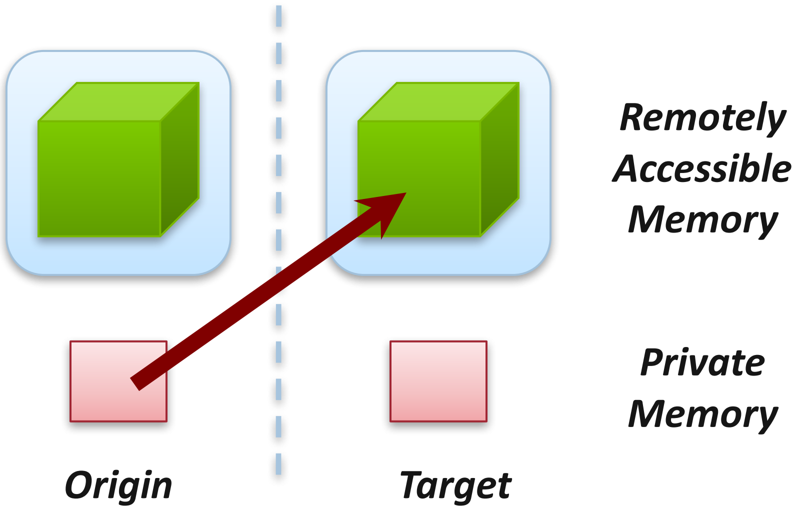
Data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - MPI_PUT
  - MPI_GET
  - MPI_ACCUMULATE **(atomic)**
  - MPI_GET_ACCUMULATE **(atomic)**
  - MPI_COMPARE_AND_SWAP **(atomic)**
  - MPI_FETCH_AND_OP **(atomic)**

Data movement: *Put*

```
MPI_Put(void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Win win)
```

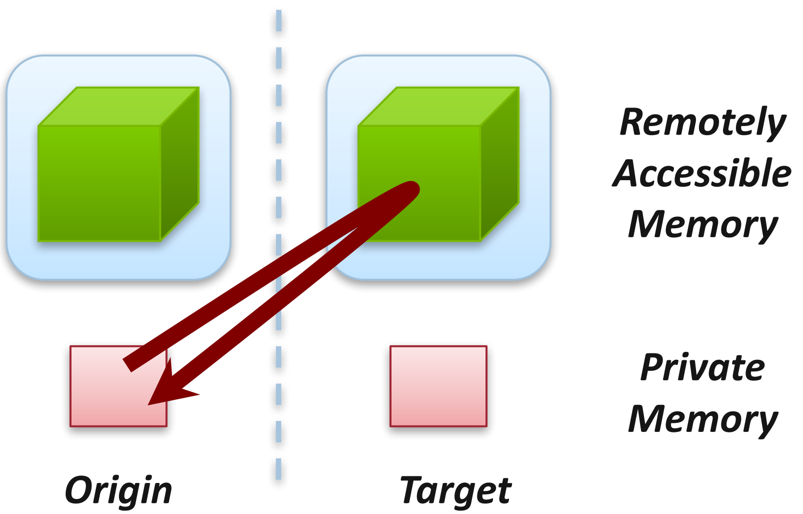
- Move data from origin, to target
- Separate data description triples for **origin** and **target**



Data movement: *Get*

```
MPI_Get(const void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Win win)
```

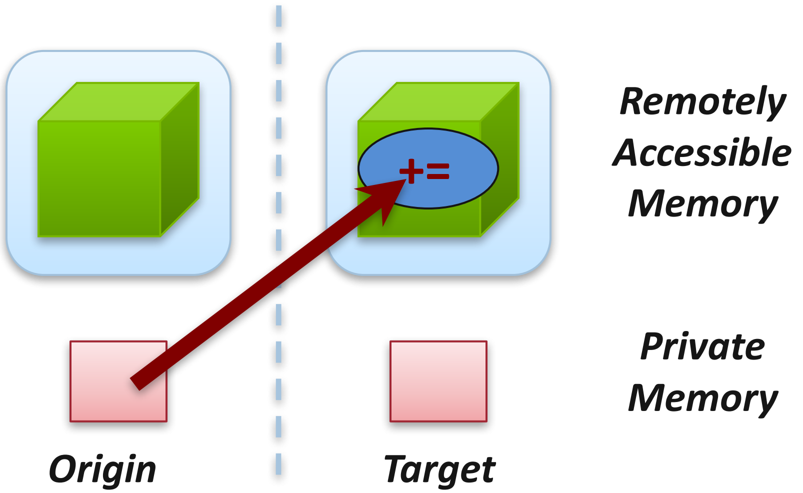
- Move data to origin, from target
- Separate data description triples for **origin** and **target**



Atomic Data Aggregation: *Accumulate*

```
MPI_Accumulate(const void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Op op, MPI_Win win)
```

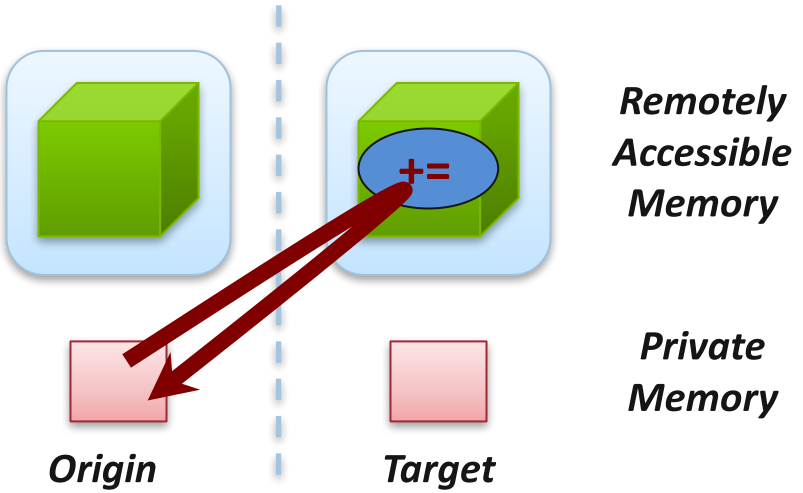
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic type elements must match
- Op = MPI_REPLACE
  - Implements *f(a,b)=b*
  - Atomic PUT



Atomic Data Aggregation: *Get Accumulate*

```
MPI_Get_accumulate(const void *origin_addr,
        int origin_count, MPI_Datatype origin_dtype,
        void *result_addr,int result_count,
        MPI_Datatype result_dtype, int target_rank,
        MPI_Aint target_disp,int target_count,
        MPI_Datatype target_dype, MPI_Op op, MPI_Win win)
```

- Atomic read-modify-write
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only
- Result stored in target buffer
- Original data stored in result buf
- Different data layouts between target/origin OK
  - Basic type elements must match
- Atomic get with MPI_NO_OP
- Atomic swap with MPI_REPLACE



Atomic Data Aggregation: *CAS and FOP*

```
MPI_Fetch_and_op(void *origin_addr, void *result_addr,
        MPI_Datatype dtype, int target_rank,
        MPI_Aint target_disp, MPI_Op op, MPI_Win win)
```

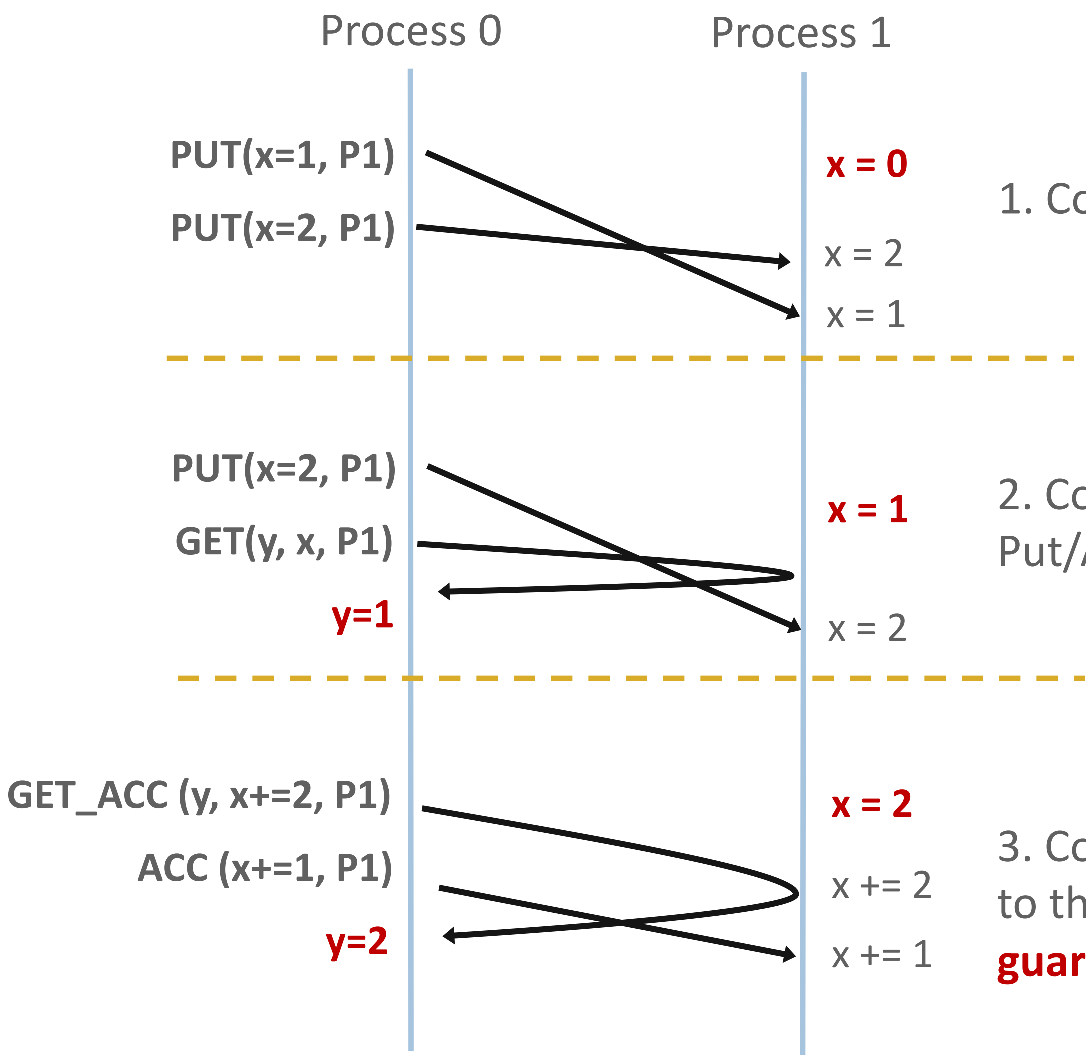
```
MPI_Compare_and_swap(void *origin_addr, void *compare_addr,
        void *result_addr, MPI_Datatype dtype, int target_rank,
        MPI_Aint target_disp, MPI_Win win)
```

- FOP: Simpler version of MPI_Get_accumulate
  - All buffers share a single predefined datatype
  - No count argument (it's always 1)
  - Simpler interface allows hardware optimization
- CAS: Atomic swap if target value is equal to compare value

Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent Put/Accumulate undefined
  - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which the occurred
  - Atomic put: Accumulate with op = MPI_REPLACE
  - Atomic get: Get_accumulate with op = MPI_NO_OP
- Accumulate operations from a given process are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW

Examples with operation ordering



1. Concurrent Puts: **undefined**

2. Concurrent Get and Put/Accumulates: **undefined**

3. Concurrent Accumulate operations to the same location : **ordering is guaranteed**

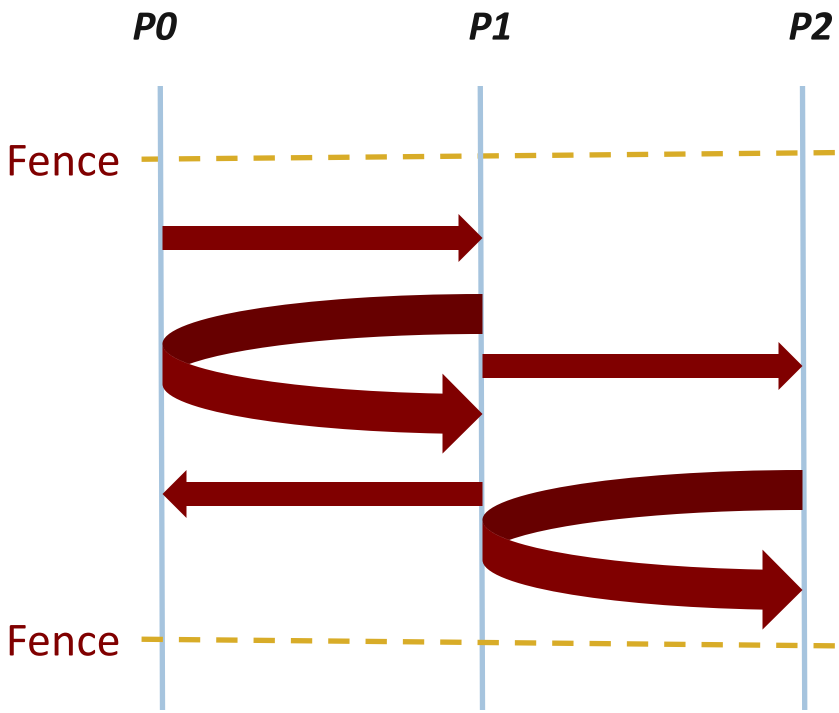
RMA Synchronization Models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics
- Three synchronization models provided by MPI:
  - Fence (active target)
  - Post-start-complete-wait (generalized active target)
  - Lock/Unlock (passive target)
- Data accesses occur within “epochs”
  - *Access epochs*: contain a set of operations issued by an origin process
  - *Exposure epochs*: enable remote processes to update a target’s window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs
    - E.g., starting, ending, and synchronizing epochs

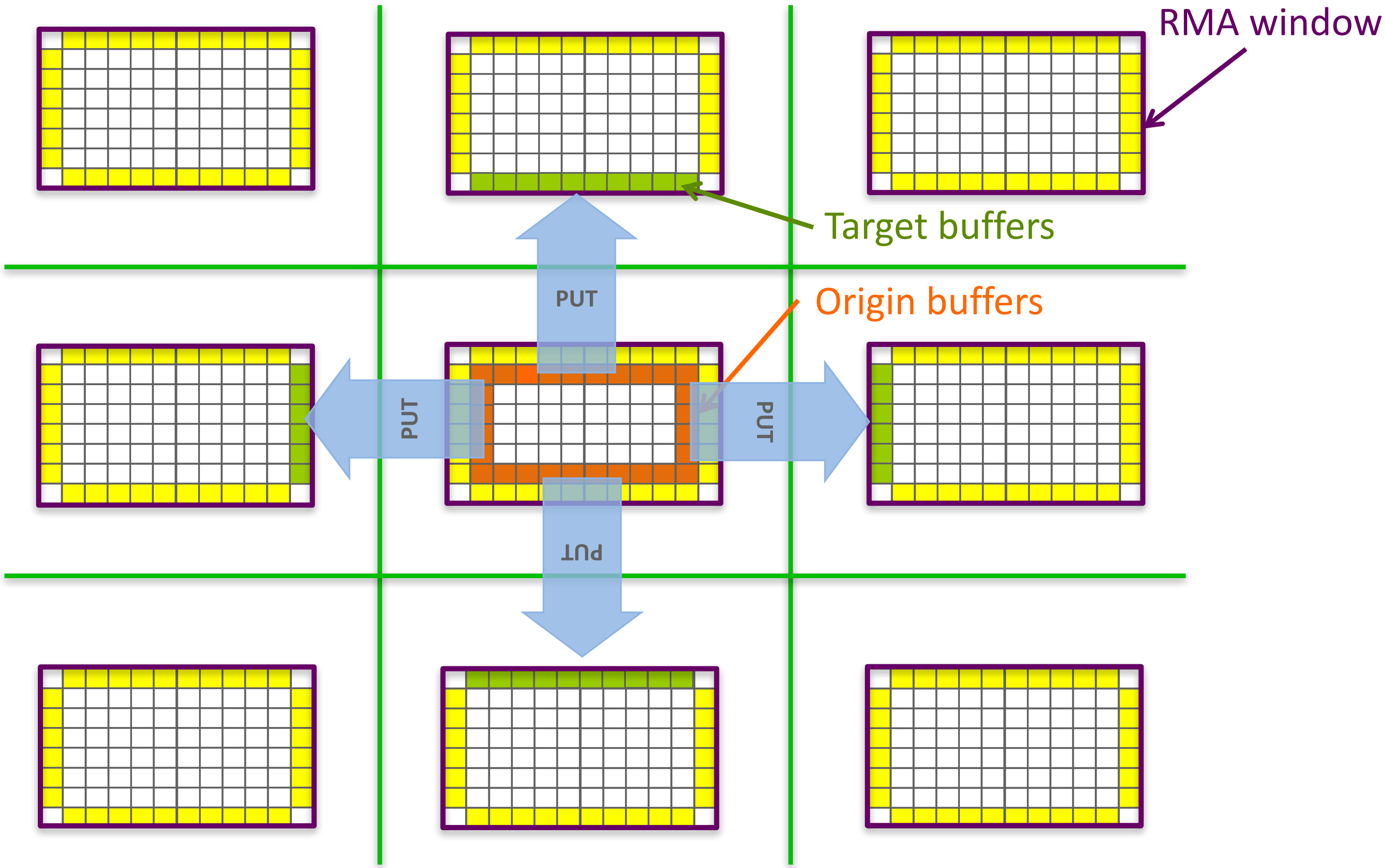
Fence: Active Target Synchronization

```
MPI_Win_fence(int assert, MPI_Win win)
```

- Collective synchronization model
- Starts *and* ends access and exposure epochs on all processes in the window
- All processes in group of “win” do an MPI_WIN_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
- All operations complete at the second fence synchronization



Implementing Stencil Computation with RMA Fence



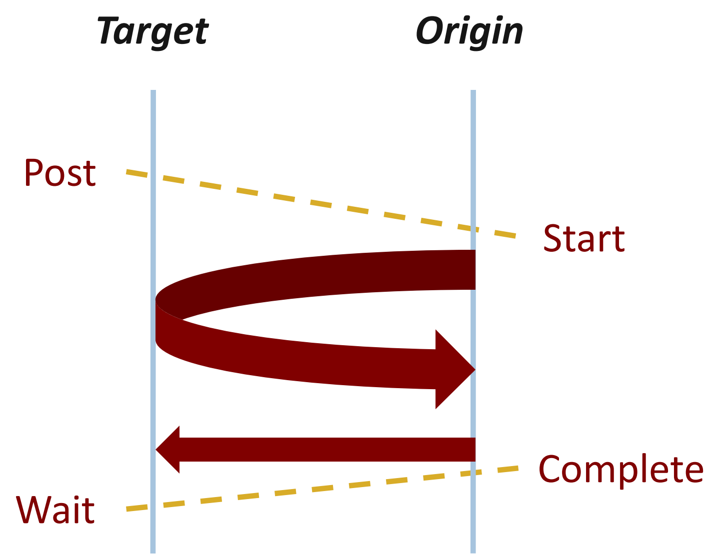
Code Example

- *stencil_mpi_ddt_rma.c*
- Use MPI_PUTs to move data, explicit receives are not needed
- Data location specified by MPI datatypes
- Manual packing of data no longer required

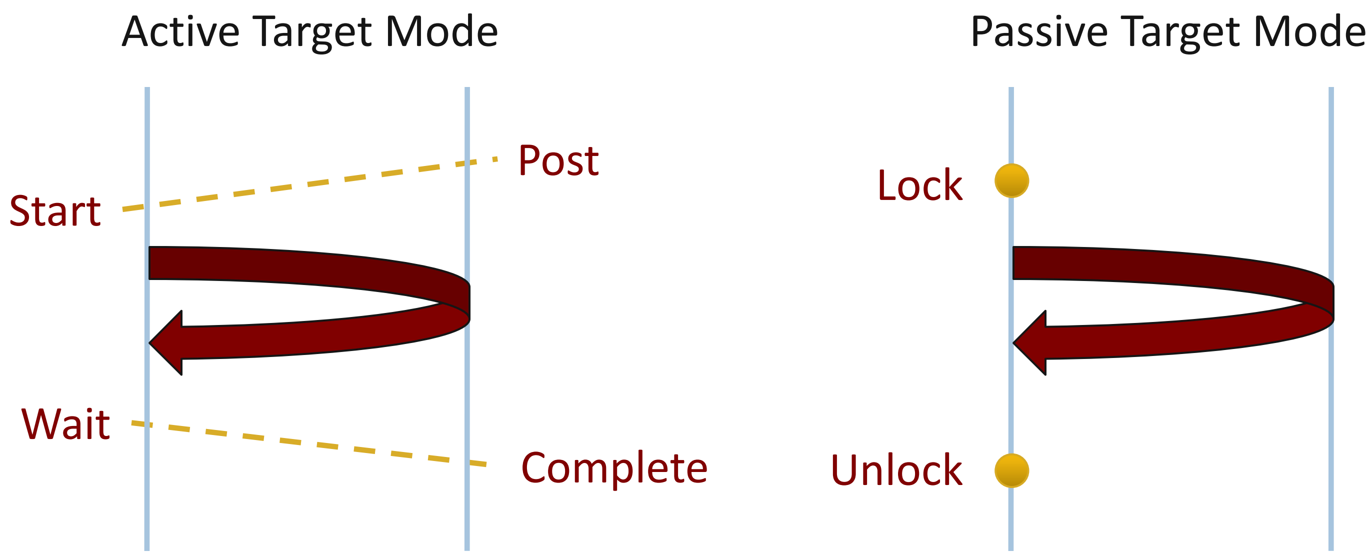
PSCW: Generalized Active Target Synchronization

```
MPI_Win_post/start(MPI_Group grp, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)
```

- Like FENCE, but origin and target specify who they communicate with
- Target: Exposure epoch
  - Opened with MPI_Win_post
  - Closed by MPI_Win_wait
- Origin: Access epoch
  - Opened by MPI_Win_start
  - Closed by MPI_Win_complete
- All synchronization operations may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets



Lock/Unlock: Passive Target Synchronization



- Passive mode: One-sided, *asynchronous* communication
  - Target does **not** participate in communication operation
- Shared memory-like model

Passive Target Synchronization

```
MPI_Win_lock(int locktype, int rank, int assert, MPI_Win win)
```

```
MPI_Win_unlock(int rank, MPI_Win win)
```

```
MPI_Win_flush/flush_local(int rank, MPI_Win win)
```

- Lock/Unlock: Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed (affects threads)
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently
- Flush: Remotely complete RMA operations to the target process
  - After completion, data can be read by target process or a different process
- Flush_local: Locally complete RMA operations to the target process

Advanced Passive Target Synchronization

```
MPI_Win_lock_all(int assert, MPI_Win win)
```

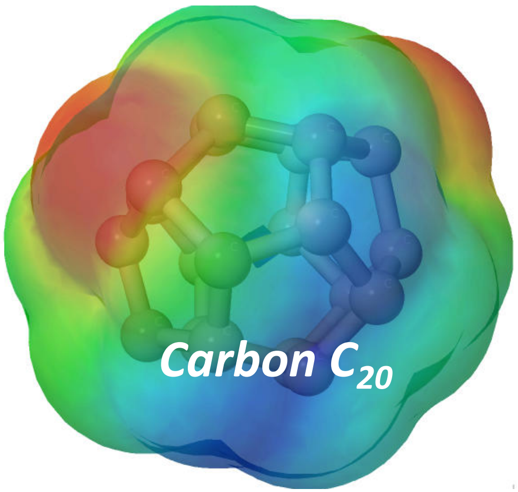
```
MPI_Win_unlock_all(MPI_Win win)
```

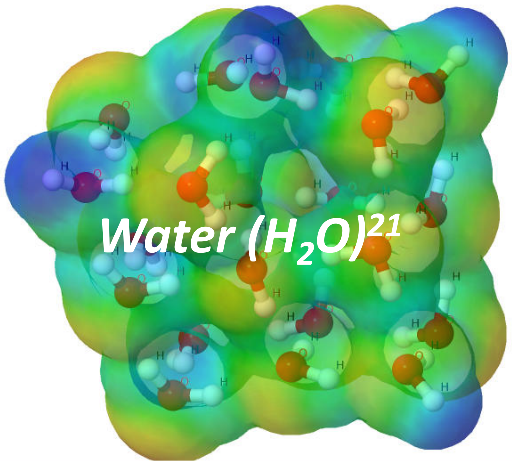
```
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```

- Lock_all: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: lock_all, put/get, flush, …, unlock_all
- Flush_all – remotely complete RMA operations to all processes
- Flush_local_all – locally complete RMA operations to all processes

NWChem [1]

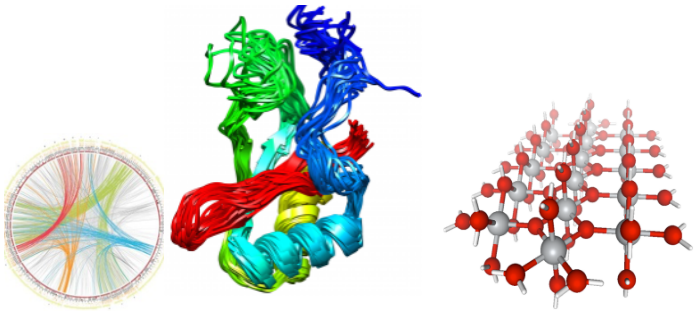
- High performance computational chemistry application suite
- Quantum level simulation of molecular systems
  - Very expensive in computation and data movement, so is used for small systems
  - Larger systems use molecular level simulations
- Composed of many simulation capabilities
  - Molecular Electronic Structure
  - Quantum Mechanics/Molecular Mechanics
  - Pseudo potential Plane-Wave Electronic Structure
  - Molecular Dynamics
- Very large code base
  - 4M LOC; Total investment of ~200M $ to date





[1] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, W.A. de Jong, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations" Comput. Phys. Commun. 181, 1477 (2010)

NWChem Communication Runtime



Applications

Global Arrays [2]

ARMCI : Communication interface for RMA[3]

ARMCI native ports

IB

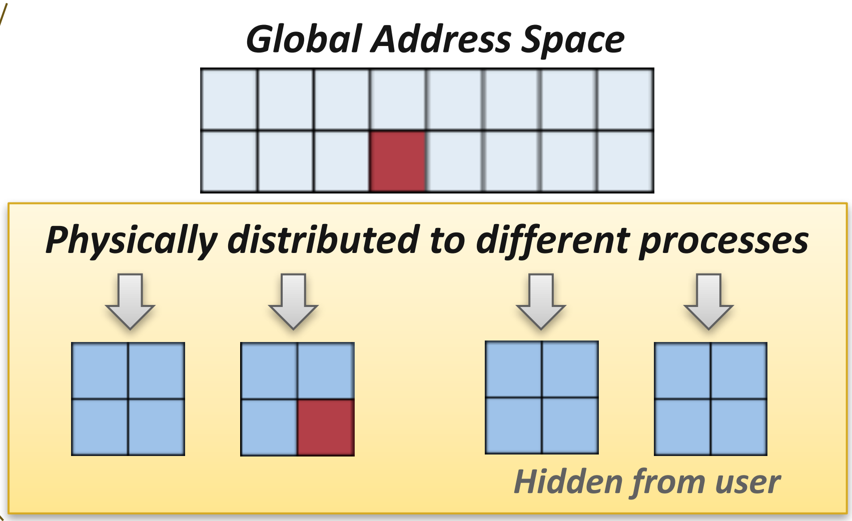
DMMAP

...

ARMCI-MPI

MPI RMA

Abstractions for distributed arrays



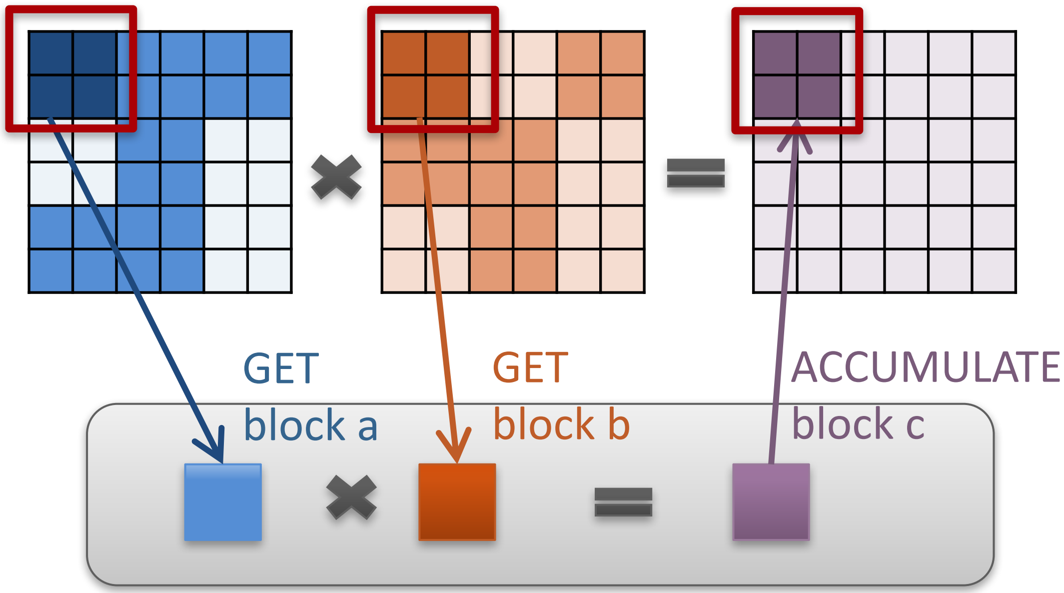
Irregularly access large amount of remote memory regions

[2] http://hpc.pnl.gov/globalarrays
[3] http://hpc.pnl.gov/armci

Get-Compute-Update

- Typical Get-Compute-Update mode in GA programming

All of the blocks are non-contiguous data



Perform DGEMM in local buffer

Pseudo code

```
for i in I blocks:
  for j in J blocks:
     for k in K blocks:
        GET block a from A
        GET block b from B
        c += a * b   /*computing*/
      end do
      ACC block c to C
      NXTASK
  end do
end do
```

Mock figure showing 2D DGEMM with block-sparse computations. In reality, NWChem uses 6D tensors.

Code Example

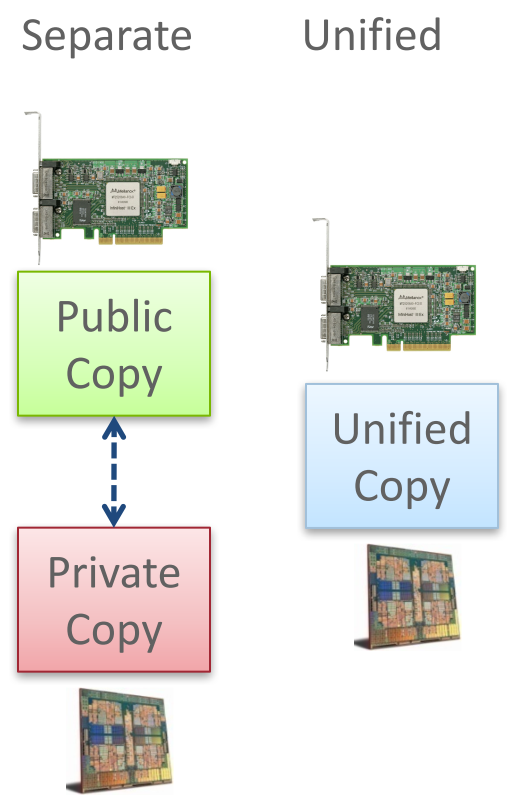
- ga_mpi_ddt_rma.c
- Only synchronization from origin processes, no synchronization from target processes

Which synchronization mode should I use, when?

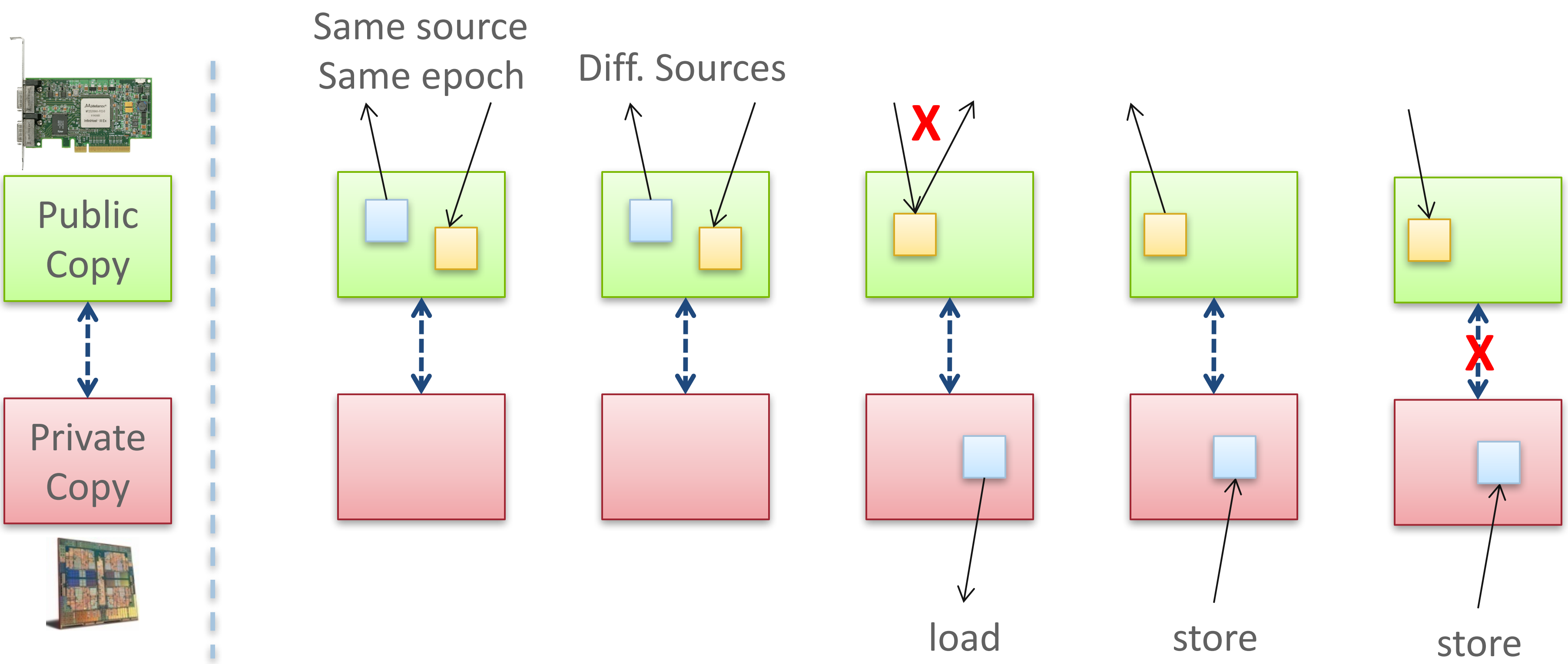
- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc…
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- Active mode: bulk synchronization
  - E.g. ghost cell exchange
- Passive mode: asynchronous data movement
  - Useful when dataset is large, requiring memory of multiple nodes
  - Also, when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays
- Passive target locking mode
  - Lock/unlock – Useful when exclusive epochs are needed
  - Lock_all/unlock_all – Useful when only shared epochs are needed

MPI RMA Memory Model

- MPI-3 provides two memory models: separate and unified
- MPI-2: Separate Model
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don't provide hardware coherence
- MPI-3: New Unified Model
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics
    - E.g. allows concurrent local/remote access
  - Provides access to full performance potential of hardware

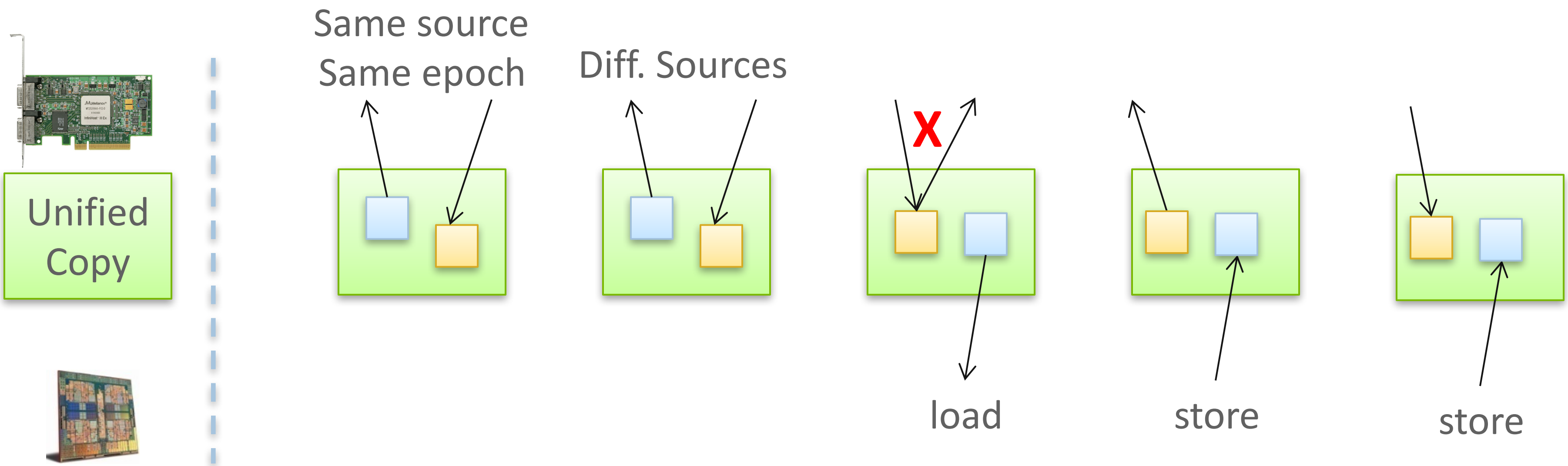


MPI RMA Memory Model (separate windows)



- Very portable, compatible with non-coherent memory systems
- Limits concurrent accesses to enable software coherence

MPI RMA Memory Model (unified windows)



- Allows concurrent local/remote accesses
- Concurrent, conflicting operations are allowed (not invalid)
  - Outcome is not defined by MPI (defined by the hardware)
- Can enable better performance by reducing synchronization

MPI RMA Operation Compatibility (Separate)

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	X	X
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	X	NOVL	NOVL	NOVL
Acc	NOVL	X	NOVL	NOVL	OVL+NOVL

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

OVL – Overlapping operations permitted

NOVL – Nonoverlapping operations permitted

X – Combining these operations is OK, but data might be garbage

MPI RMA Operation Compatibility (Unified)

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	NOVL	NOVL
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	NOVL	NOVL	NOVL	NOVL
Acc	NOVL	NOVL	NOVL	NOVL	OVL+NOVL

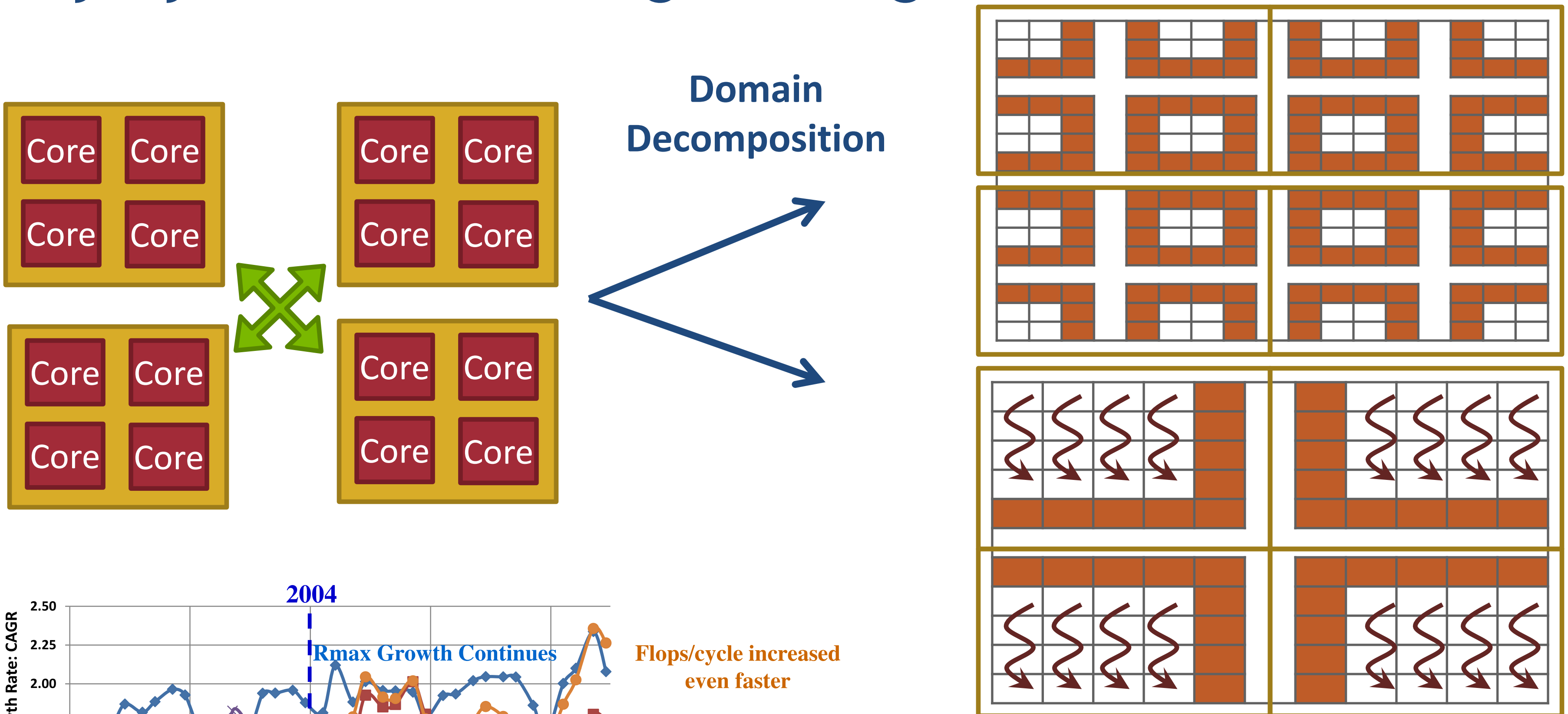
This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

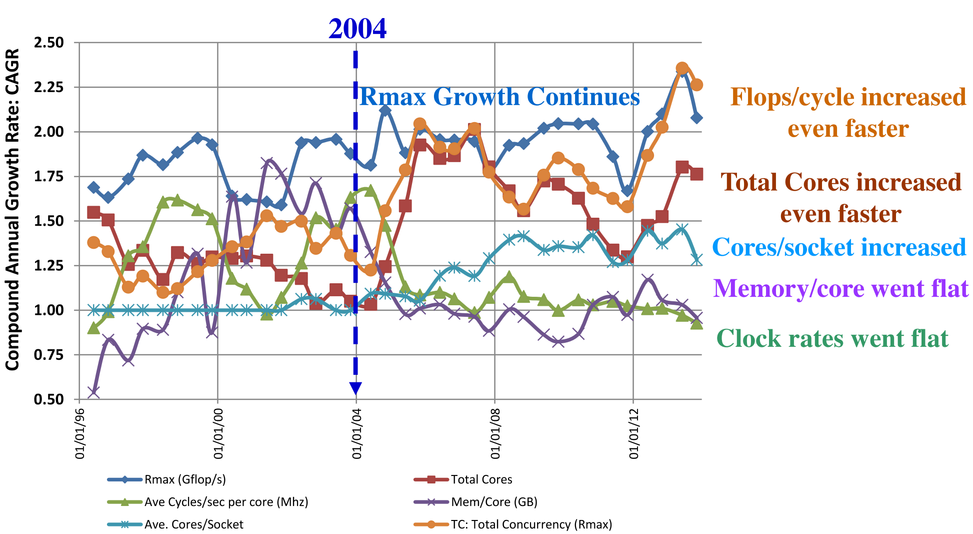
OVL – Overlapping operations permitted

NOVL – Nonoverlapping operations permitted

Hybrid Programming with Threads, Shared Memory, and GPUs

Why Hybrid MPI + X Programming?





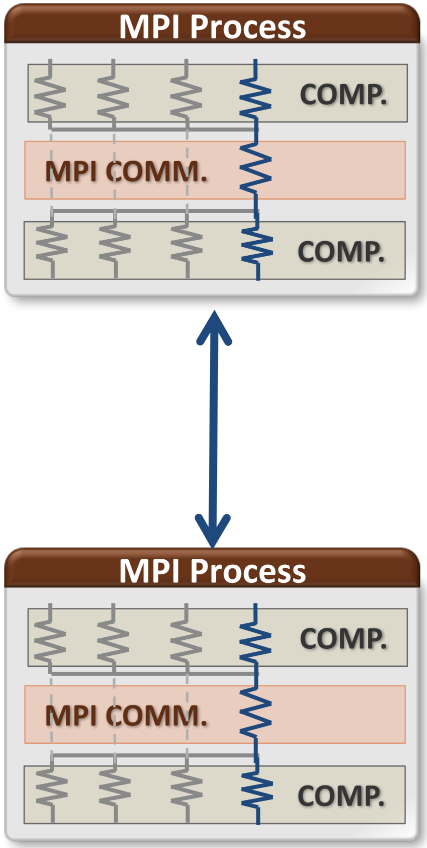
Growth of node resources in the Top500 systems. Peter Kogge: "Reading the Tea-Leaves: How Architecture Has Evolved at the High End". IPDPS 2014 Keynote

- Sharing promotes cooperation
  – Reduced memory consumption
  – Efficient use of shared resources: caches, TLB entries, network endpoints, etc.

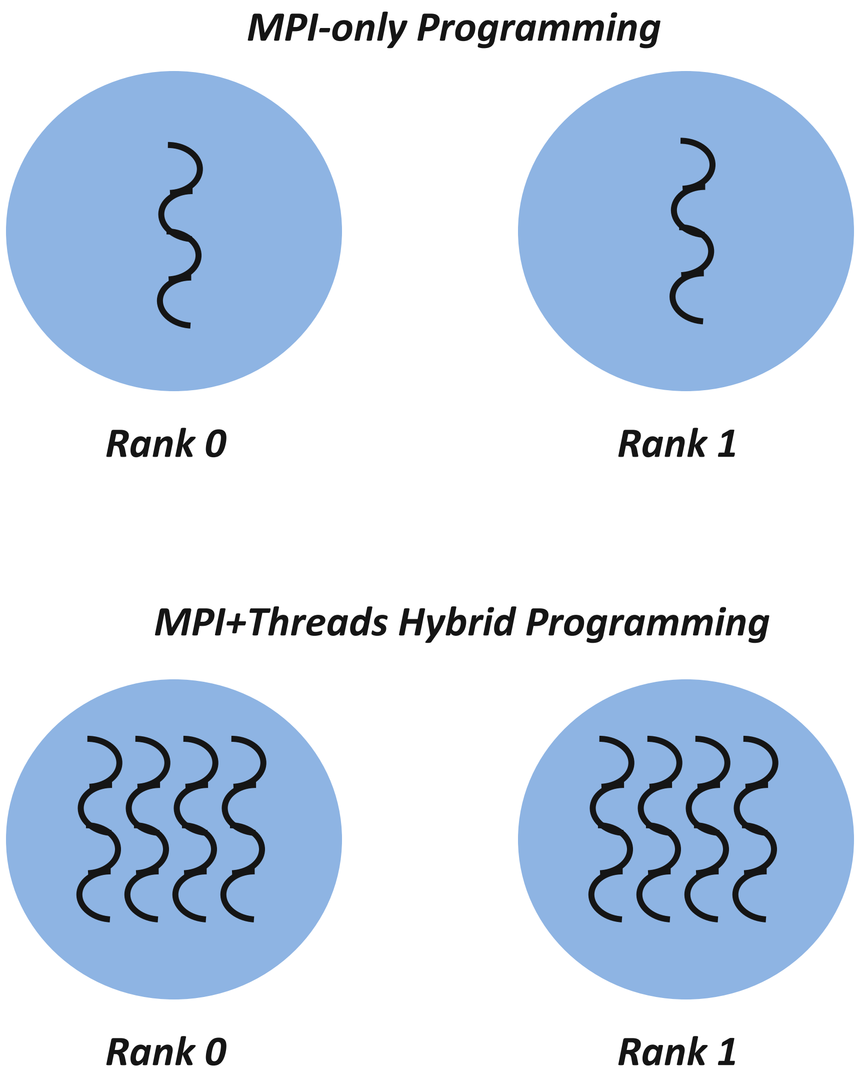
MPI + Threads

MPI and Threads

- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.



Hybrid Programming with MPI+Threads



- In MPI-only programming, each MPI process has a single thread of execution
- In MPI+threads hybrid programming, there can be multiple threads executing simultaneously
  - All threads share all MPI objects (communicators, requests)
  - The MPI implementation might need to take precautions to make sure the state of the MPI stack is consistent

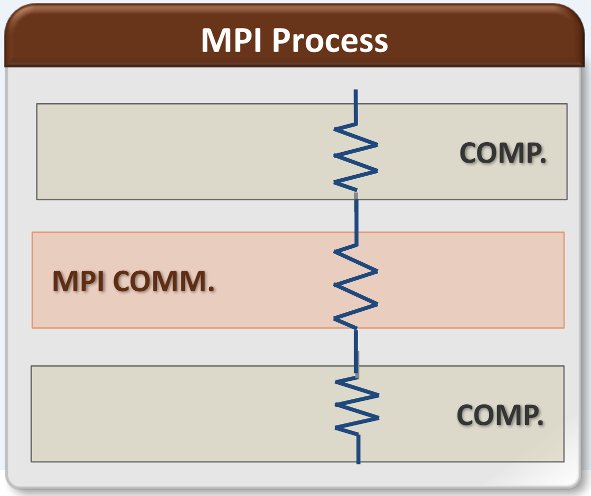
MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
  - MPI_THREAD_SINGLE: only one thread exists in the application
  - MPI_THREAD_FUNNELED: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init_thread)
  - MPI_THREAD_SERIALIZED: multithreaded, but only one thread *at a time* makes MPI calls
  - MPI_THREAD_MULTIPLE: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZED
- MPI defines an alternative to MPI_Init
  - MPI_Init_thread(requested, provided)
    - *Application specifies level it needs; MPI implementation returns level it supports*

MPI_THREAD_SINGLE

- There are no additional user threads in the system
  - E.g., there are no OpenMP parallel regions

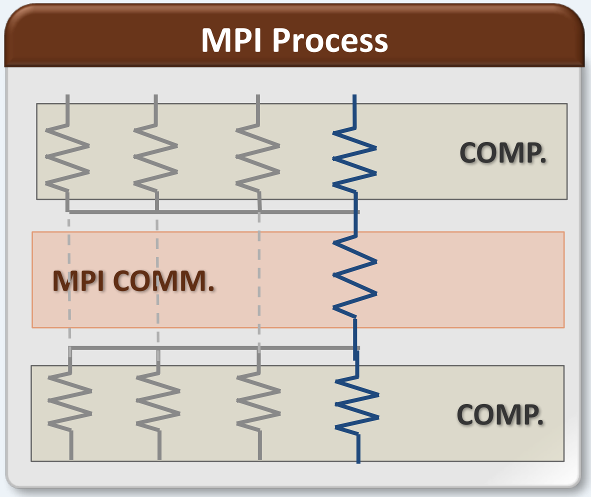
```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```



MPI_THREAD_FUNNELED

- All MPI calls are made by the **master** thread
  - Outside the OpenMP parallel regions
  - In OpenMP master regions

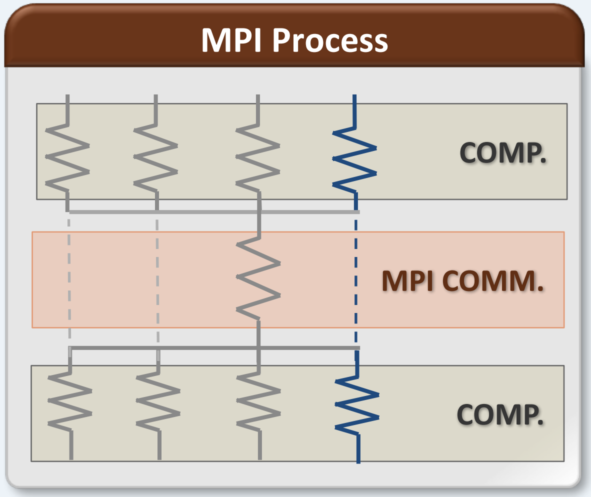
```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED) MPI_Abort(MPI_COMM_WORLD,1);

#pragma omp parallel for
    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();
    return 0;
}
```



MPI_THREAD_SERIALIZED

- Only **one** thread can make MPI calls at a time
  - Protected by OpenMP critical regions

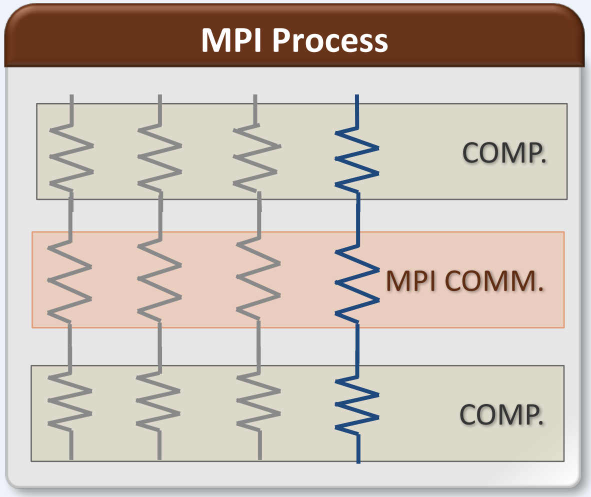
```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    if (provided < MPI_THREAD_SERIALIZED) MPI_Abort(MPI_COMM_WORLD,1);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
#pragma omp critical
        /* Do MPI stuff */
    }

    MPI_Finalize();
    return 0;
}
```



MPI_THREAD_MULTIPLE

- **Any** thread can make MPI calls any time (restrictions apply)

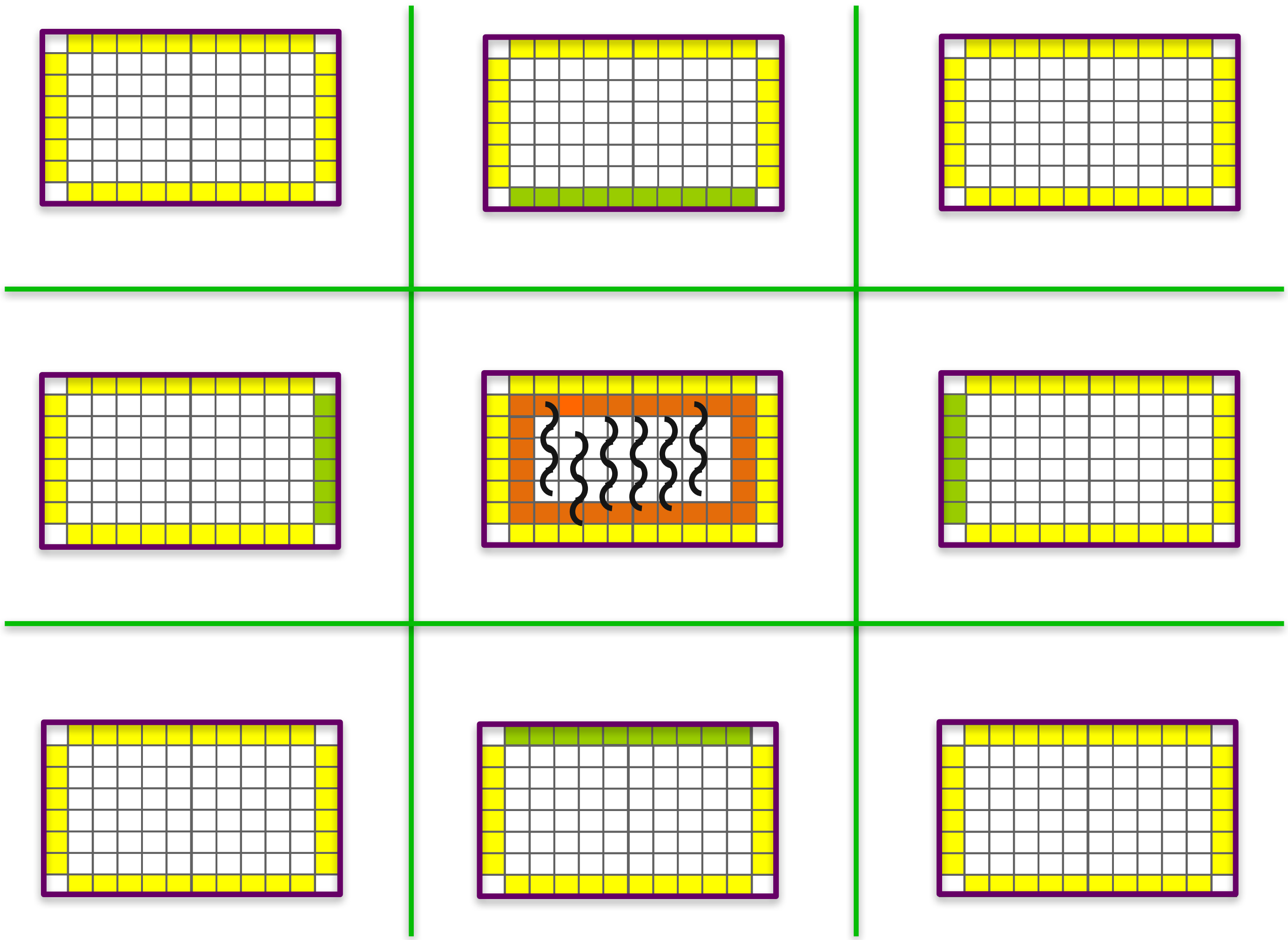
```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_MULTIPLE) MPI_Abort(MPI_COMM_WORLD,1);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }

    MPI_Finalize();
    return 0;
}
```



Threads and MPI

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support MPI_THREAD_MULTIPLE
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported
  - MPI Standard *mandates* MPI_THREAD_SINGLE for MPI_Init
- *A threaded MPI program that does not call MPI_Init_thread is an incorrect program (common user error we see)*

Implementing Stencil Computation using MPI_THREAD_FUNNELED



Code Examples

- *stencil_mpi_ddt_funneled.c*
- Parallelize computation (OpenMP parallel for)
- Main thread does all communication

MPI Semantics and MPI_THREAD_MULTIPLE

- ***Ordering:*** When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
  - Ordering is maintained within each thread
  - User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
    - E.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator
  - It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
    - E.g., accessing an info object from one thread and freeing it from another thread
- ***Progress:*** Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives

	Process 0	Process 1
Thread 0	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 1	MPI_Barrier(comm)	MPI_Barrier(comm)

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives

Process 0		*Process 1*	
Thread 1	Thread 2	Thread 1	Thread 2
MPI_Bcast(comm)			MPI_Barrier(comm)
	MPI_Barrier(comm)	MPI_Bcast(comm)	

- P0 and P1 can have different orderings of Bcast and Barrier
- Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with RMA

```
int main(int argc, char ** argv)
{
    /* Initialize MPI and RMA window */

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        target = rand();
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target, 0, win);
        MPI_Put(..., win);
        MPI_Win_unlock(target, win);
    }

    /* Free MPI and RMA window */

    return 0;
}
```

Different threads can lock the same process causing multiple locks to the same target before the first lock is unlocked

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Object Management

Process 0

Thread 1	Thread 2
	MPI_Comm_free(comm)
MPI_Bcast(comm)	

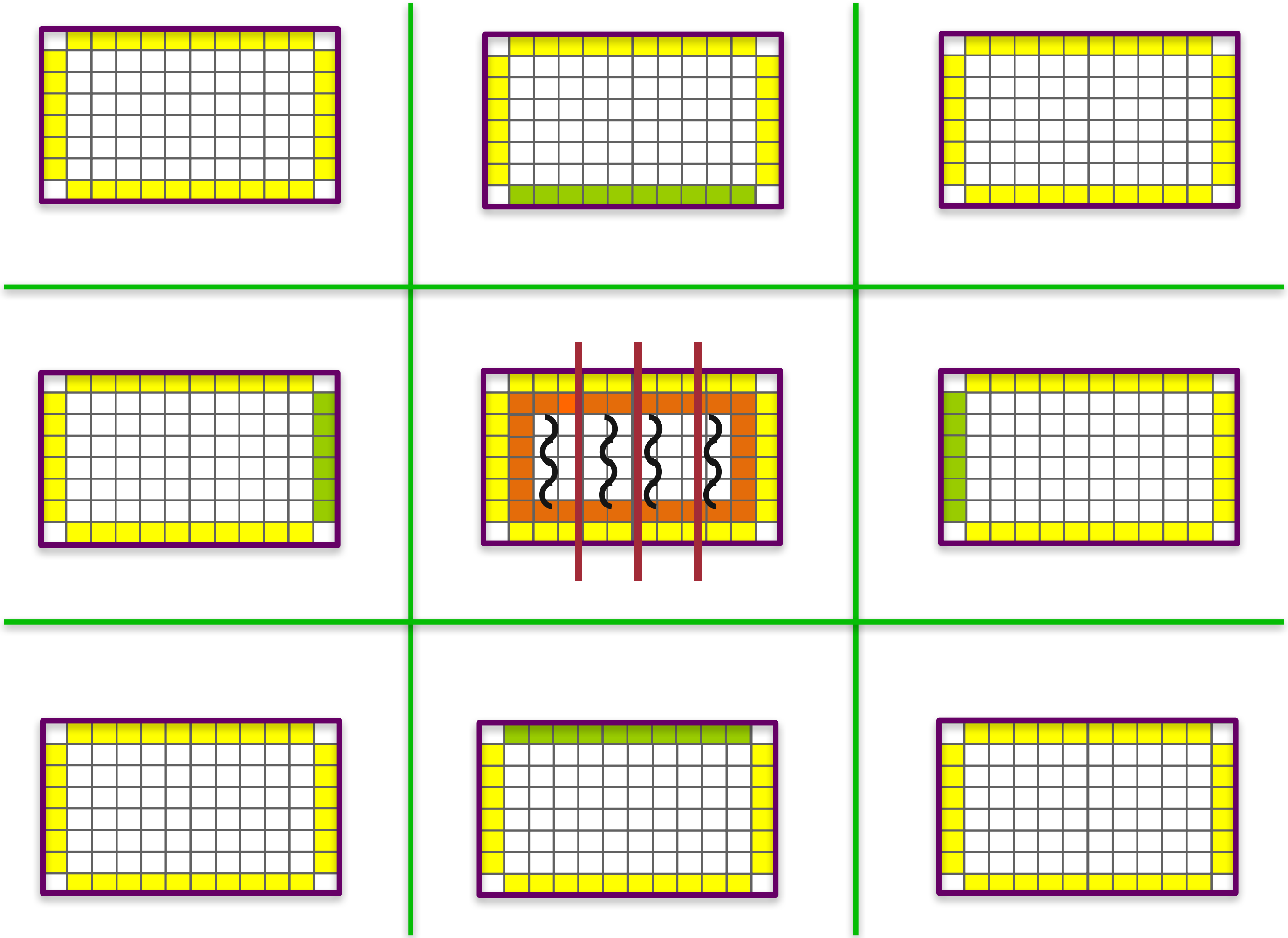
- The user has to make sure that one thread is not using an object while another thread is freeing it
  - This is essentially an ordering issue; the object might get freed before it is used

Blocking Calls in MPI_THREAD_MULTIPLE: Correct Example

	Process 0	Process 1
Thread 1	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 2	MPI_Send(dst=1)	MPI_Send(dst=0)

- An implementation must ensure that the above example never deadlocks for any ordering of thread execution
- That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress.

Implementing Stencil Computation using MPI_THREAD_MULTIPLE



Code Examples

- *stencil_mpi_ddt_multiple.c*
- Divide the process memory among OpenMP threads
- Each thread responsible for communication and computation

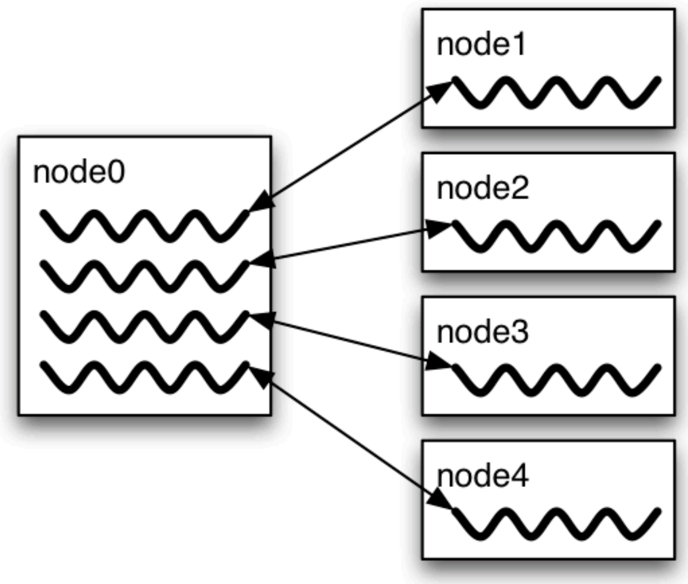
The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE
- They probably support MPI_THREAD_FUNNELED even if they don’t admit it.
  - Does require thread-safety for some system routines (e.g. malloc)
  - On most systems `-pthread` will guarantee it (OpenMP implies `-pthread`)
- Many (but not all) implementations support THREAD_MULTIPLE
  - Hard to implement efficiently though (thread synchronization issues)
- Bulk-synchronous OpenMP programs (loops parallelized with OpenMP, communication between loops) only need FUNNELED
  - So don’t need “thread-safe” MPI for many hybrid programs
  - But watch out for Amdahl’s Law!

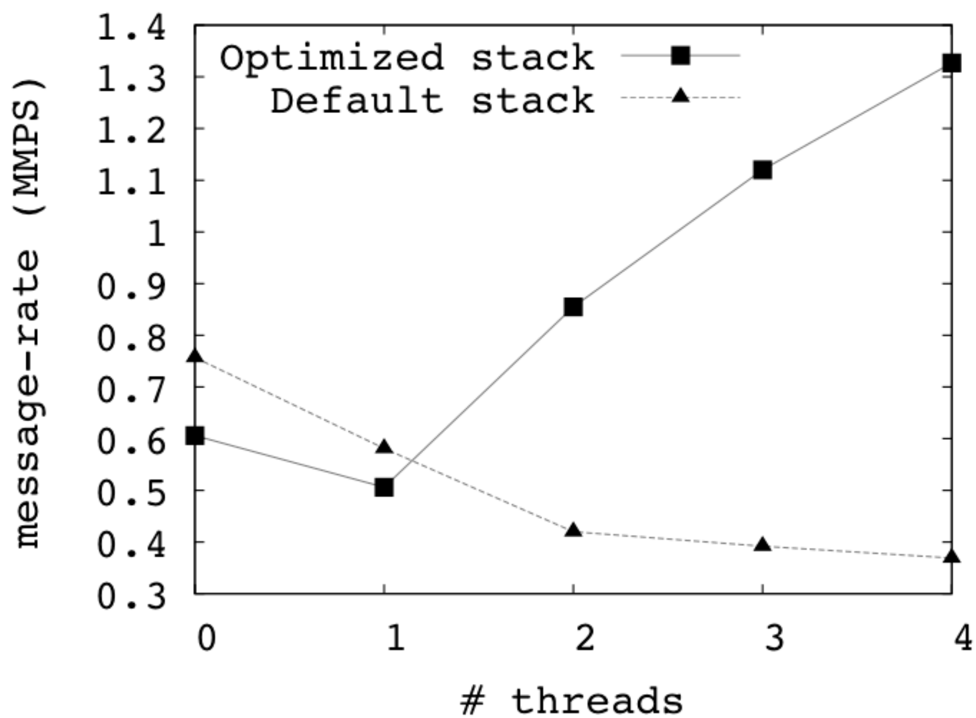
Performance with MPI_THREAD_MULTIPLE

- Thread safety does not come for free
- The implementation must access/modify several shared objects (e.g. message queues) in a consistent manner
- To measure the performance impact, we ran tests to measure communication performance when using multiple threads versus multiple processes
  - For results, see Thakur/Gropp paper: “Test Suite for Evaluating Performance of Multithreaded MPI Communication,” *Parallel Computing*, 2009

Message Rate Results on BG/P



Message Rate Benchmark



"Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems" EuroMPI 2010

Why is it hard to optimize MPI_THREAD_MULTIPLE

- MPI internally maintains several resources
- Because of MPI semantics, it is required that all threads have access to some of the data structures
  - E.g., thread 1 can post an `Irecv`, and thread 2 can wait for its completion – thus the request queue has to be shared between both threads
  - Since multiple threads are accessing this shared queue, thread-safety is required to ensure a consistent state of the queue – adds a lot of overhead

Hybrid Programming: Correctness Requirements

- Hybrid programming with MPI+threads does not do much to reduce the complexity of thread programming
  - Your application still has to be a correct multi-threaded application
  - On top of that, you also need to make sure you are correctly following MPI semantics
- Many commercial debuggers offer support for debugging hybrid MPI+threads applications (mostly for MPI+Pthreads and MPI+OpenMP)

An Example we encountered

- We received a bug report about a very simple multithreaded MPI program that hangs
- Run with 2 processes
- Each process has 2 threads
- Both threads communicate with threads on the other process as shown in the next slide
- We spent several hours trying to debug MPICH before discovering that the bug is actually in the user’s program ☹

2 Proceses, 2 Threads, Each Thread Executes this Code

```
for (j = 0; j < 2; j++) {
    if (rank == 1) {
        for (i = 0; i < 2; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        for (i = 0; i < 2; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
    }
    else {  /* rank == 0 */
        for (i = 0; i < 2; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
        for (i = 0; i < 2; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
}
```

Intended Ordering of Operations

Rank 0	***Rank 1***
2 recvs (T1)	2 sends (T1)
2 sends (T1)	2 recvs (T1)
2 recvs (T1)	2 sends (T1)
2 sends (T1)	2 recvs (T1)
2 recvs (T2)	2 sends (T2)
2 sends (T2)	2 recvs (T2)
2 recvs (T2)	2 sends (T2)
2 sends (T2)	2 recvs (T2)

- Every send matches a receive on the other rank

Possible Ordering of Operations in Practice

Rank 0		***Rank 1***	
2 recvs (T1)		2 sends (T1)	
2 sends (T1)		1 recv (T1)	
1 recv (T1)			2 sends (T2)
	1 recv (T2)		1 recv (T2)
1 recv (T1)	1 recv (T2)	1 recv (T1)	1 recv (T2)
2 sends (T1)	2 sends (T2)	2 sends (T1)	2 sends (T2)
	2 recvs (T2)	2 recvs (T1)	2 recvs (T2)
	2 sends (T2)		

- Because the MPI operations can be issued in an arbitrary order across threads, all threads could block in a RECV call

Some Things to Watch for in OpenMP

- Limited thread and no explicit memory affinity control (but see OpenMP 4.0 and 4.5)
  - "First touch" (have intended "owning" thread perform first access) provides initial static mapping of memory
    - Next touch (move ownership to most recent thread) could help
  - No portable way to reassign memory affinity – reduces the effectiveness of OpenMP when used to improve load balancing.
- Memory model can require explicit "memory flush" operations
  - Defaults allow race conditions
  - Humans notoriously poor at recognizing all races
    - It only takes one mistake to create a hard-to-find bug

Some Things to Watch for in MPI + OpenMP

- No interface for apportioning resources between MPI and OpenMP
  - On an SMP node, how many MPI processes and how many OpenMP Threads?
    - Note the static nature assumed by this question
  - Note that having more threads than cores can be important for hiding latency
    - Requires very lightweight threads
- Competition for resources
  - Particularly memory bandwidth and network access
  - Apportionment of network access between threads and processes is also a problem, as we've already seen.

Where Does the MPI + OpenMP Hybrid Model Work Well?

- Compute-bound loops
  - Many operations per memory load
- Fine-grain parallelism
  - Algorithms that are latency-sensitive
- Load balancing
  - Similar to fine-grain parallelism; ease of
- Memory bound loops

Compute-Bound Loops

- Loops that involve many operations per load from memory
  - This can happen in some kinds of matrix assembly, for example.
  - Jacobi update not compute bound

Fine-Grain Parallelism

- Algorithms that require frequent exchanges of small amounts of data
- E.g., in blocked preconditioners, where fewer, larger blocks, each managed with OpenMP, as opposed to more, smaller, single-threaded blocks in the all-MPI version, gives you an algorithmic advantage (e.g., fewer iterations in a preconditioned linear solution algorithm).
- Even if memory bound

Load Balancing

- Where the computational load isn't exactly the same in all threads/processes; this can be viewed as a variation on fine-grained access.
- OpenMP schedules can handle some of this
  - For very fine grain cases, a mix of static and dynamic scheduling may be more efficient
  - Current research looking at more elaborate and efficient schedules for this case

Memory-Bound Loops

- Where read data is shared, so that cache memory can be used more efficiently.
- Example: Table lookup for evaluating equations of state
  - Table can be shared
  - If table evaluated as necessary, evaluations can be shared

Where is Pure MPI Better?

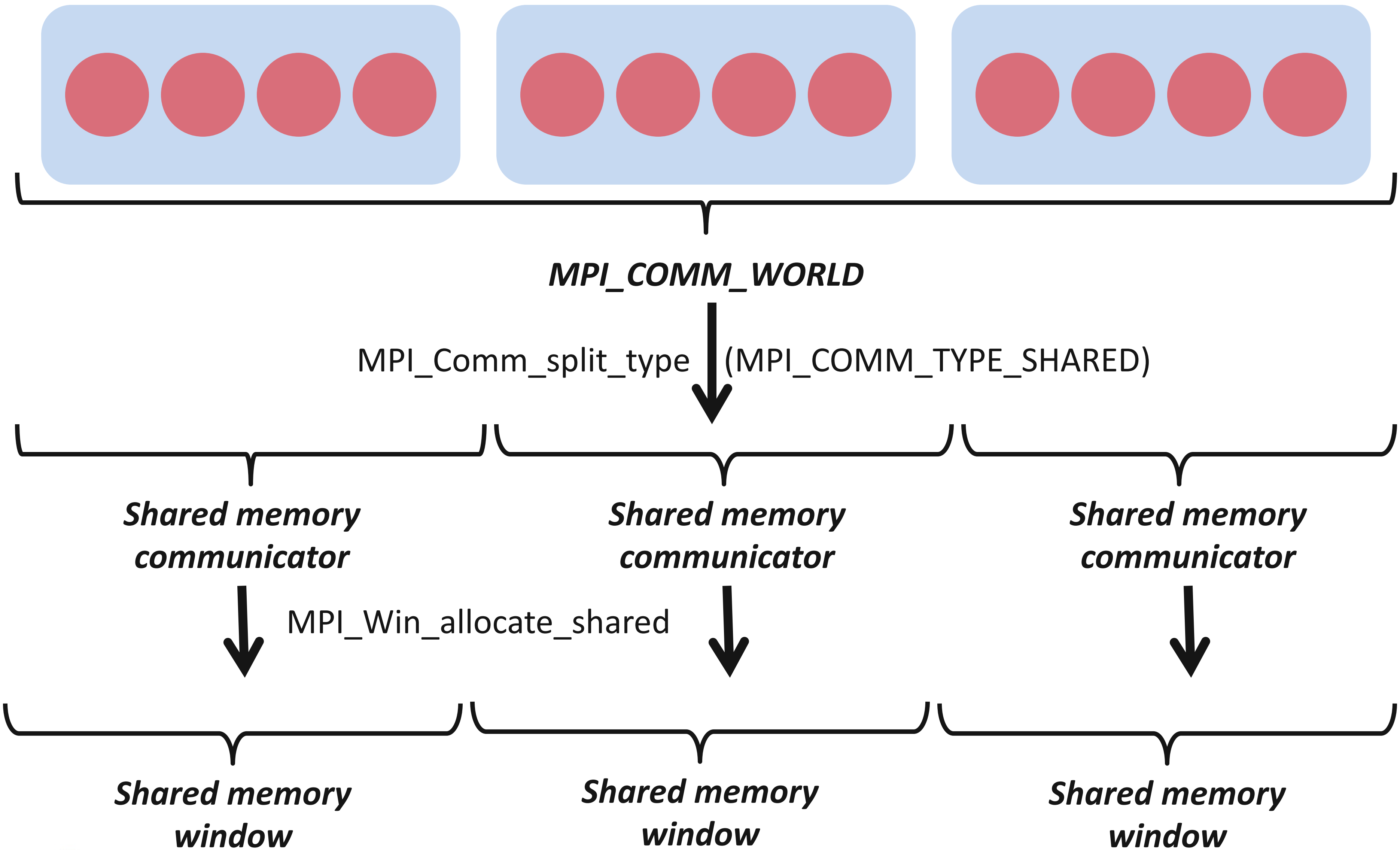
- Trying to use OpenMP + MPI on very regular, memory-bandwidth-bound computations is likely to lose because of the better, programmer-enforced memory locality management in the pure MPI version.
- Another reason to use more than one MPI process - if a single process (or thread) can't saturate the interconnect - then use multiple communicating processes or threads.
  - Note that threads and processes are not equal

MPI + Shared-Memory

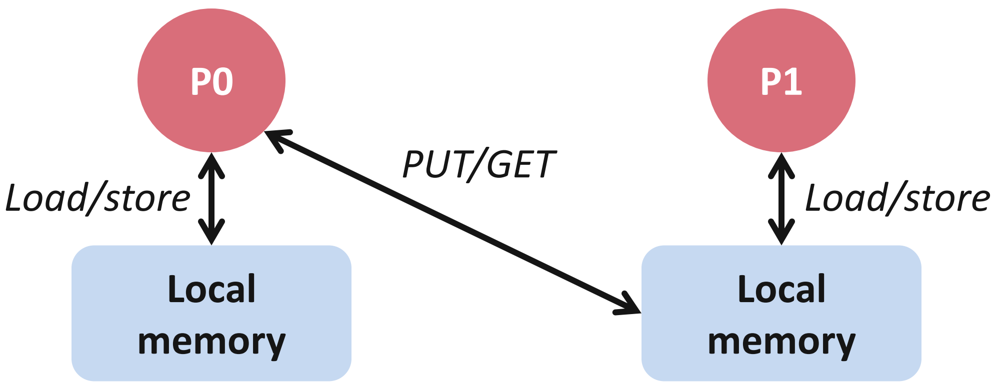
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads

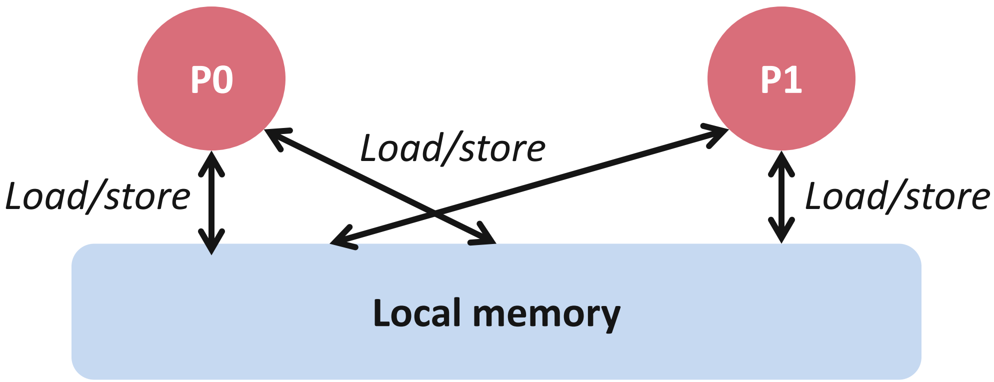
Creating Shared Memory Regions in MPI



Regular RMA windows vs. Shared memory windows



Traditional RMA windows



Shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - E.g., x[100] = 10
- All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations
- Can be very useful when processes want to use threads only to get access to all of the memory on the node
  - You can create a shared memory window and put your shared data

MPI_COMM_SPLIT_TYPE

```
MPI_Comm_split_type(MPI_Comm comm, int split_type,
                    int key, MPI_Info info, MPI_Comm *newcomm)
```

- Create a communicator where processes "share a property"
  - Properties are defined by the "split_type"
- Arguments:
  - comm - input communicator (handle)
  - Split_type - property of the partitioning (integer)
  - Key - Rank assignment ordering (nonnegative integer)
  - info - info argument (handle)
  - newcomm- output communicator (handle)

MPI_WIN_ALLOCATE_SHARED

```
MPI_Win_allocate_shared(MPI_Aint size, int disp_unit,
               MPI_Info info, MPI_Comm comm, void *baseptr,
               MPI_Win *win)
```

- Create a remotely accessible memory region in an RMA window
  - Data exposed in a window can be accessed with RMA ops or load/store
- Arguments:
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
  - baseptr - pointer to exposed local data
  - win - window (handle)

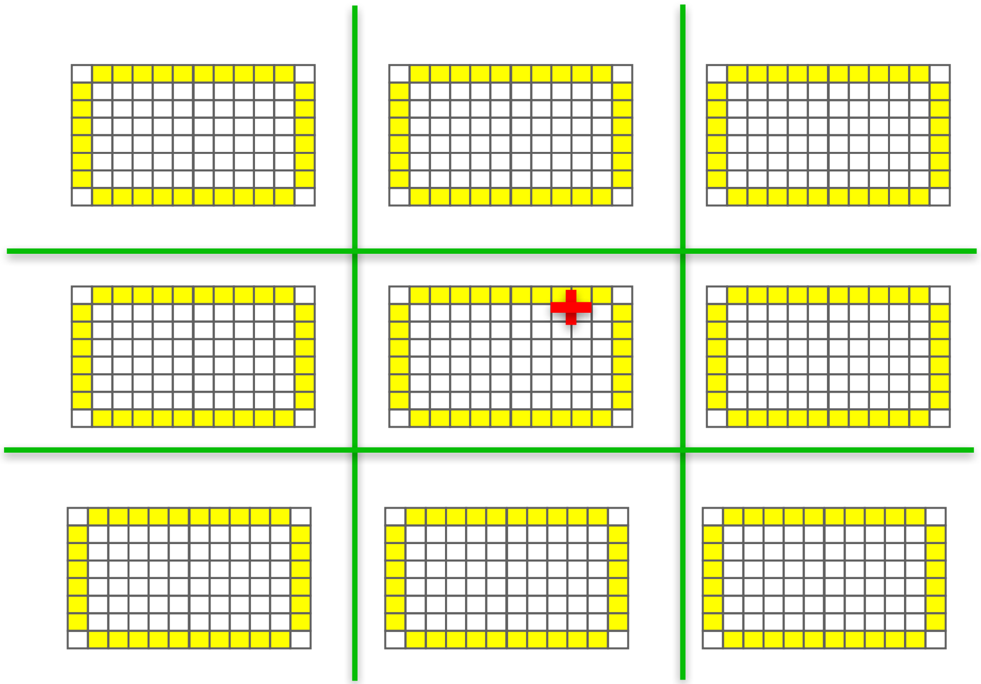
Shared Arrays with Shared memory windows

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Win_sync(win);

    /* use shared memory */

    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

Memory allocation and placement

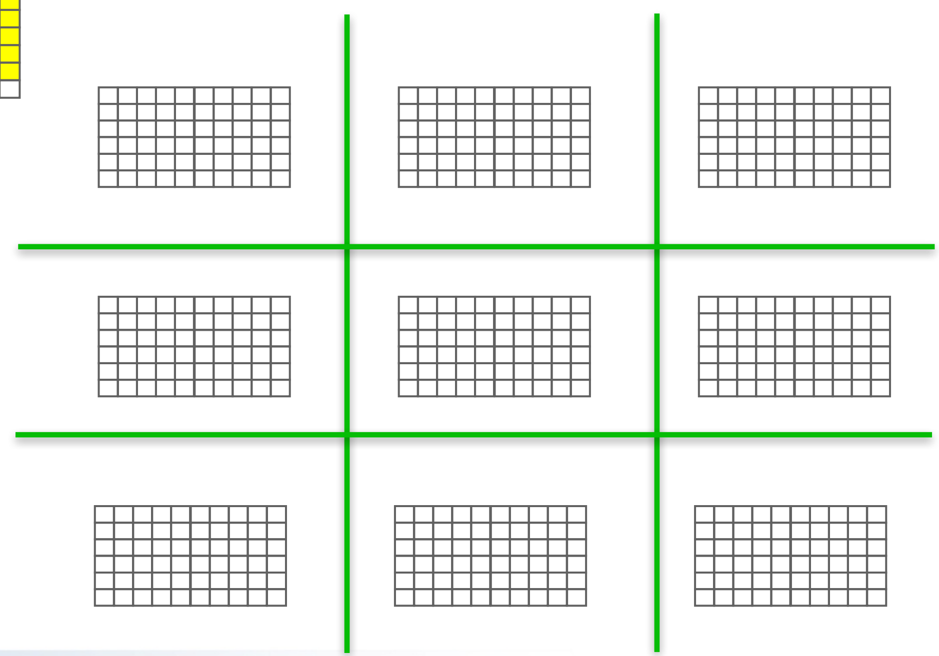
- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed (e.g., which physical memory it will be pinned to)
  - Implementations can choose their own strategies, though it is expected that an implementation will try to place shared memory allocated by a process "close to it"
- The total allocated shared memory on a communicator is contiguous by default
  - Users can pass an info hint called "noncontig" that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement

Example Computation: Stencil



Message passing model requires ghost-cells to be explicitly communicated to neighbor processes

In the shared-memory model, there is no communication. Neighbors directly access your data.



Walkthrough of 2D Stencil Code with Shared Memory Windows

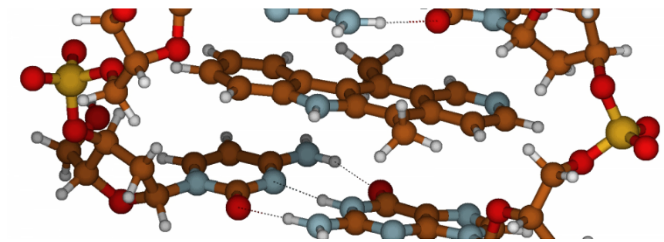
- *stencil_mpi_shmem.c*

Which Hybrid Programming Method to Adopt?

- It depends on the application, target machine, and MPI implementation
- When should I use process shared memory?
  - The only resource that needs sharing is memory
  - Few allocated objects need sharing (easy to place them in a public shared region)
- When should I use threads?
  - More than memory resources need sharing (e.g., TLB)
  - Many application objects require sharing
  - Application computation structure can be easily parallelized with high-level OpenMP loops

Example: Quantum Monte Carlo

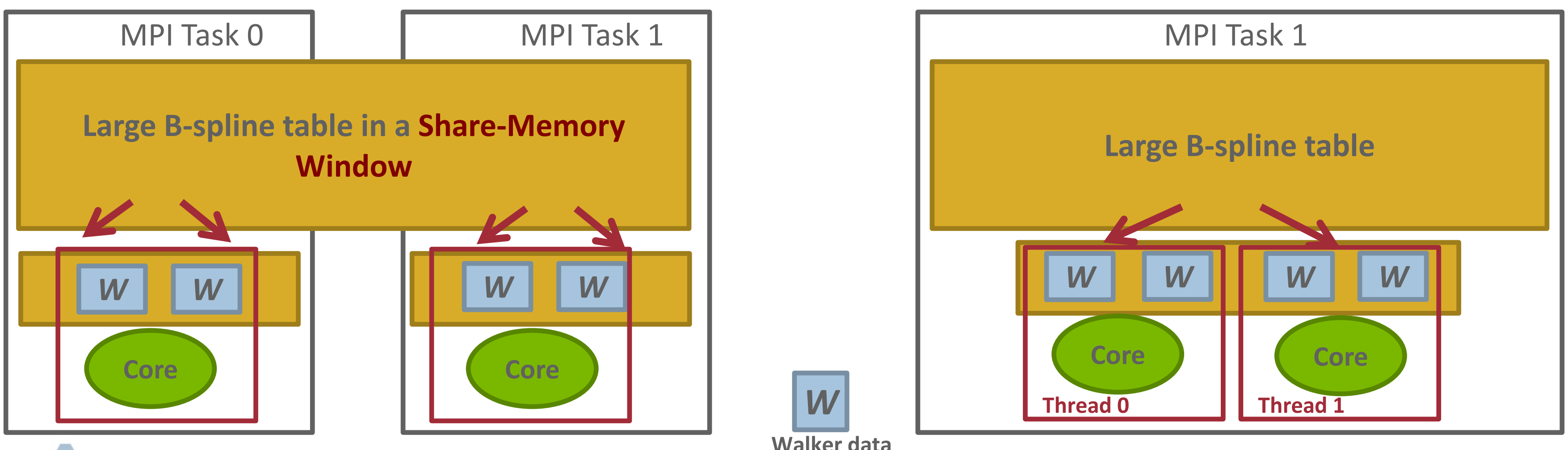
QMCPACK



- Memory capacity bound with MPI-only
- Hybrid approaches
  - MPI + threads (e.g. X = OpenMP, Pthreads)
  - MPI + shared-memory (X = MPI)
- Can use direct load/store operations instead of message passing

MPI + Shared-Memory (MPI 3.0)
- **Everything private by default**
- **Expose shared data explicitly**

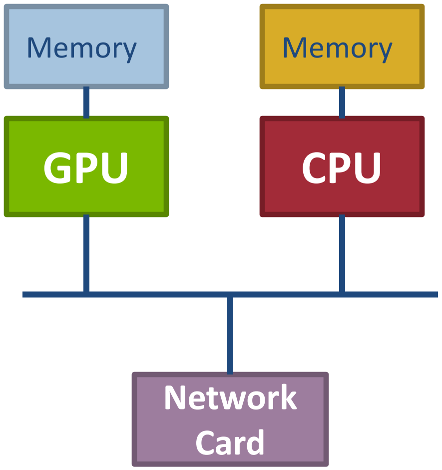
MPI + Threads
- **Share everything by default**
- **Privatize data when necessary**



MPI + Accelerators

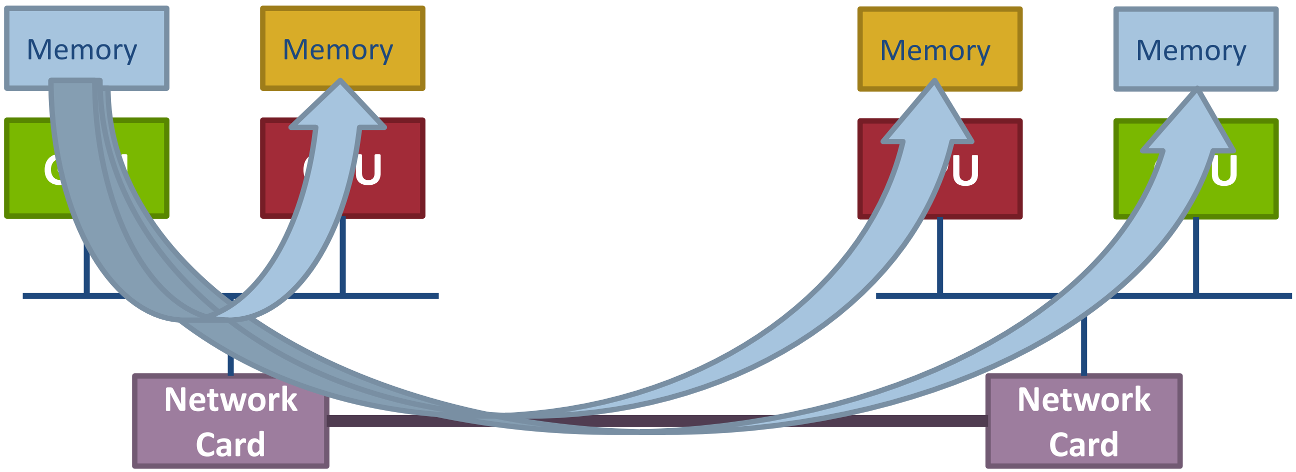
Accelerators in Parallel Computing

- General purpose, highly parallel processors
  - High FLOPs/Watt and FLOPs/$
  - Unit of execution *Kernel*
  - Separate memory subsystem
  - Programming Models: OpenAcc, CUDA, OpenCL, …
- Clusters with accelerators are becoming common
- New programmability and performance challenges for programming models and runtime systems



MPI + Accelerator Programming Examples (1/2)

FAQ: How to move data between GPUs with MPI?



MPI + Accelerator Programming Examples (2/2)

CUDA

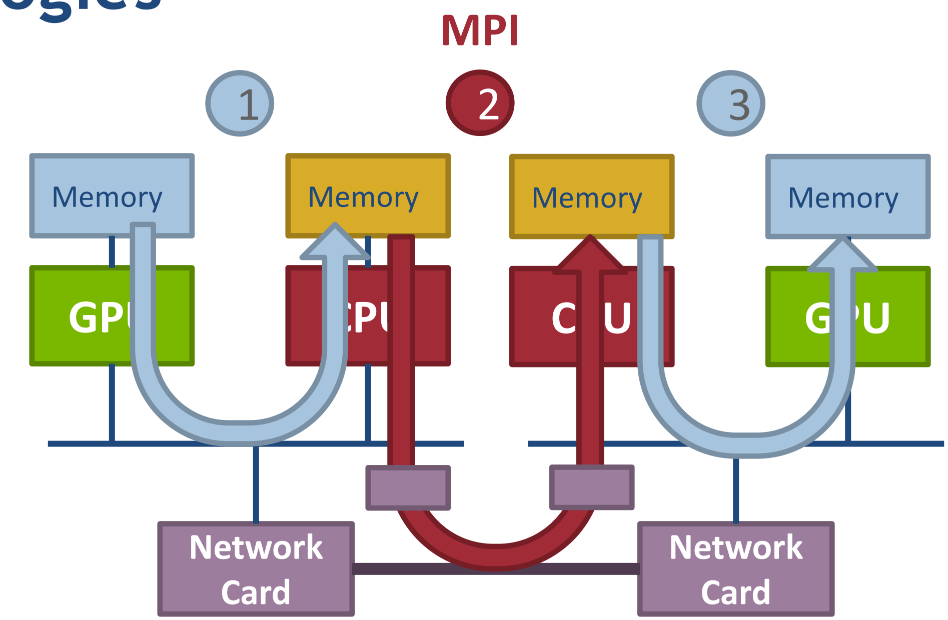
```
double *dev_buf, *host_buf;
cudaMalloc(&dev_buf, size);
cudaMallocHost(&host_buf, size);

if(my_rank == sender) {
    computation_on_GPU(dev_buf);
    cudaMemcpy(host_buf, dev_buf, size, …);
    MPI_Isend(host_buf, size, …);
} else {
    MPI_Irecv(host_buf, size, …);
    cudaMemcpy(dev_buf, host_buf, size, …);
    computation_on_GPU(dev_buf);
}
```

OpenACC

```
double *buf;
buf = (double*)malloc(size * sizeof(double));
#pragma acc enter data create(buf[0:size])

if(my_rank == sender) {
    computation_on_GPU(buf);
    #pragma acc update host (buf[0:size])
    MPI_Isend(buf, size, …);
} else {
    MPI_Irecv(buf, size, …);
    #pragma acc update device (buf[0:size])
    computation_on_GPU(buf);
}
```

MPI with Old GPU Technologies

- MPI only sees host memory
- User has to ensure data copies on host and device are updated consistently
- Several memory copy operations are required
- No overlapping between device-host memory transfers and network communication
- **No MPI optimization opportunities**



Network pinned/registered buffer

CUDA

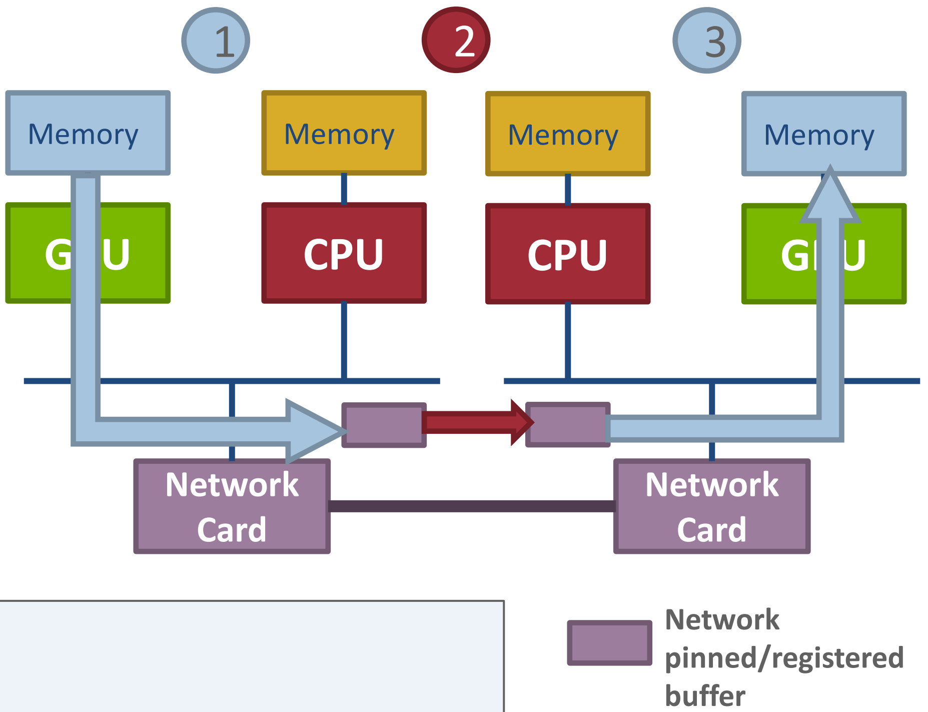
```
computation_on_GPU(dev_buf);
cudaMemcpy(host_buf, dev_buf, size, …);
MPI_Isend(host_buf, size, …);

MPI_Irecv(host_buf, size, …);
cudaMemcpy(dev_buf, host_buf, size, …);
computation_on_GPU(dev_buf);
```

OpenACC

```
computation_on_GPU(buf);
#pragma acc update host (buf[0:size])
MPI_Isend(buf, size, …);

MPI_Irecv(buf, size, …);
#pragma acc update device (buf[0:size])
computation_on_GPU(buf);
```

MPI with Unified Virtual Addressing (UVA)

- The same virtual address space for all processors, host or device (e.g., CUDA >= 4)
- User can pass device pointer to MPI
- MPI implementation needs to query for the owner (host or device) of the data
- If data on device, **the MPI implementation can optimize** as follows:
  - Reduce the number of memory copies and DMA operations
  - Do better overlapping of data transfers



CUDA

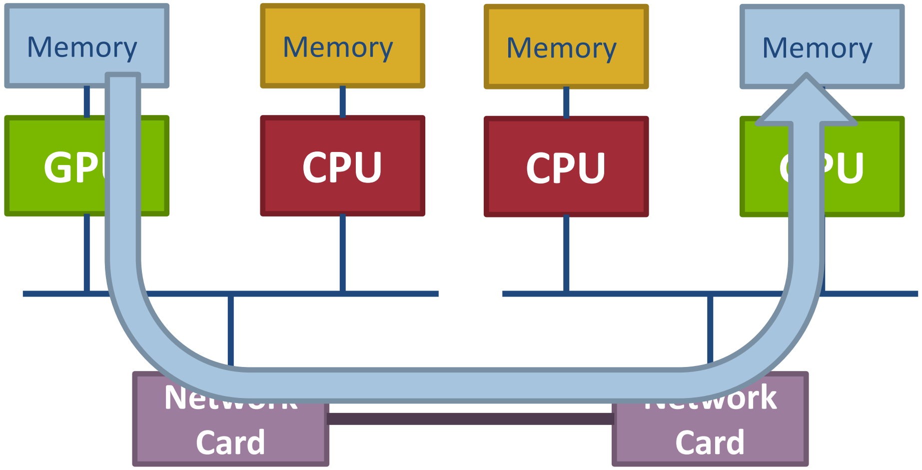
```
computation_on_GPU(dev_buf);
MPI_Isend(dev_buf, size, …);


MPI_Irecv(dev_buf, size, …);
computation_on_GPU(dev_buf);
```

OpenACC

```
computation_on_GPU(buf);
#pragma acc host_data use_device (buf)
MPI_Isend(buf, size, …);

#pragma acc host_data use_device (buf)
MPI_Irecv(buf, size, …);
computation_on_GPU(buf);
```

MPI with UVA + GPUDirect

- The hardware supports direct GPU-to-GPU data transfers within or across nodes
- MPI implementations may use the following **optimizations** to transfer data between GPUs
  - Can use directly GPU memory for RDMA communication
  - Peer-to-peer data transfers when GPUs are on the same node



CUDA

```
computation_on_GPU(dev_buf);
MPI_Isend(dev_buf, size, …);


MPI_Irecv(dev_buf, size, …);
computation_on_GPU(dev_buf);
```

OpenACC

```
computation_on_GPU(buf);
#pragma acc host_data use_device (buf)
MPI_Isend(buf, size, …);

#pragma acc host_data use_device (buf)
MPI_Irecv(buf, size, …);
computation_on_GPU(buf);
```

Advanced Topics: Nonblocking Collectives, Topologies, and Neighborhood Collectives

Nonblocking Collective Communication

- Nonblocking (send/recv) communication
  - Deadlock avoidance
  - Overlapping communication/computation
- Collective communication
  - Collection of pre-defined optimized routines
- → Nonblocking collective communication
  - Combines both techniques (more than the sum of the parts ☺)
  - System noise/imbalance resiliency
  - Semantic advantages

Nonblocking Collective Communication

- Nonblocking variants of all collectives
  - MPI_Ibcast(<bcast args>, MPI_Request *req);
- Semantics
  - Function returns no matter what
  - No guaranteed progress (quality of implementation)
  - Usual completion calls (wait, test) + mixing
  - Out-of order completion
- Restrictions
  - No tags, in-order matching
  - Send and vector buffers may not be updated during operation
  - MPI_Cancel not supported
  - No matching with blocking collectives

Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI

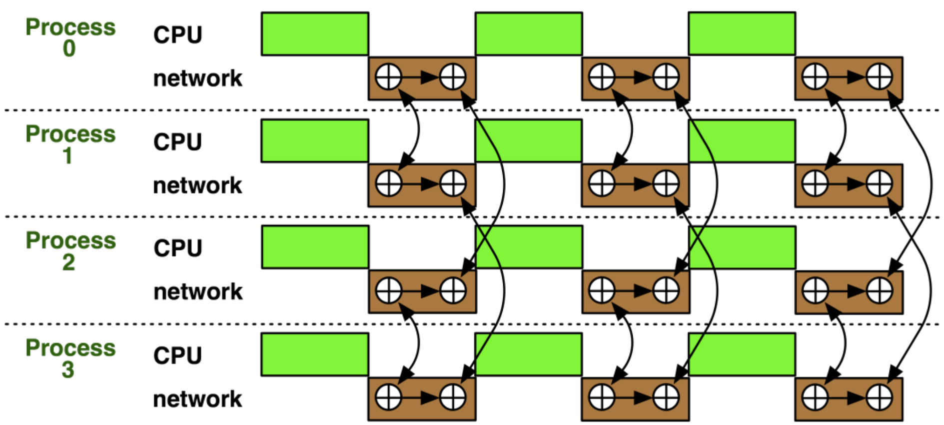
Nonblocking Collective Communication

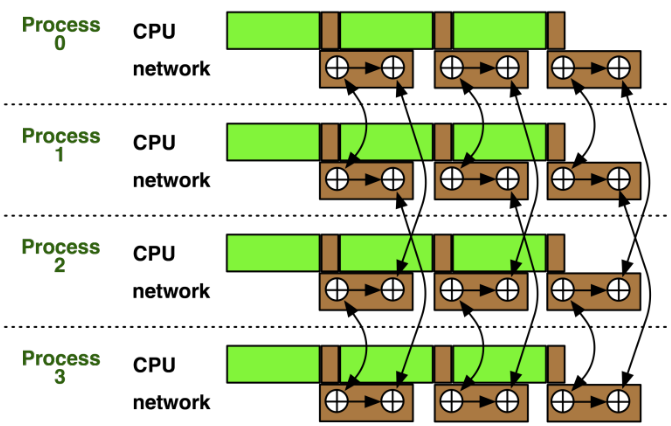
- Semantic advantages
  - Enable asynchronous progression (and manual)
    - Software pipelining
  - Decouple data transfer and synchronization
    - Noise resiliency!
  - Allow overlapping communicators
    - See also neighborhood collectives
  - Multiple outstanding operations at any time
    - Enables pipelining window

Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI

Nonblocking Collectives Overlap

- Software pipelining
  - More complex parameters
  - Progression issues
  - Not scale-invariant





Hoefler: Leveraging Non-blocking Collective Communication in High-performance Applications

A Non-Blocking Barrier?

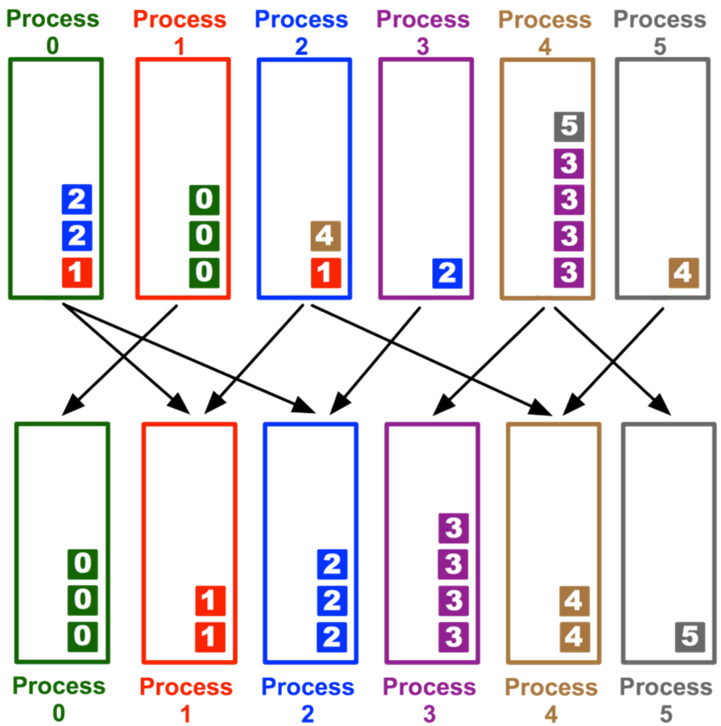
- What can that be good for? Well, quite a bit!
- Semantics:
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary
- Uses:
  - Overlap barrier latency (small benefit)
  - Use the split semantics! Processes **notify** non-collectively but **synchronize** collectively!

A Semantics Example: DSDE

- Dynamic Sparse Data Exchange
  - Dynamic: comm. pattern varies across iterations
  - Sparse: number of neighbors is limited (O(log P))
  - Data exchange: only senders know neighbors
- Main Problem: metadata
  - Determine who wants to send how much data to me

  (I must post receive and reserve memory)

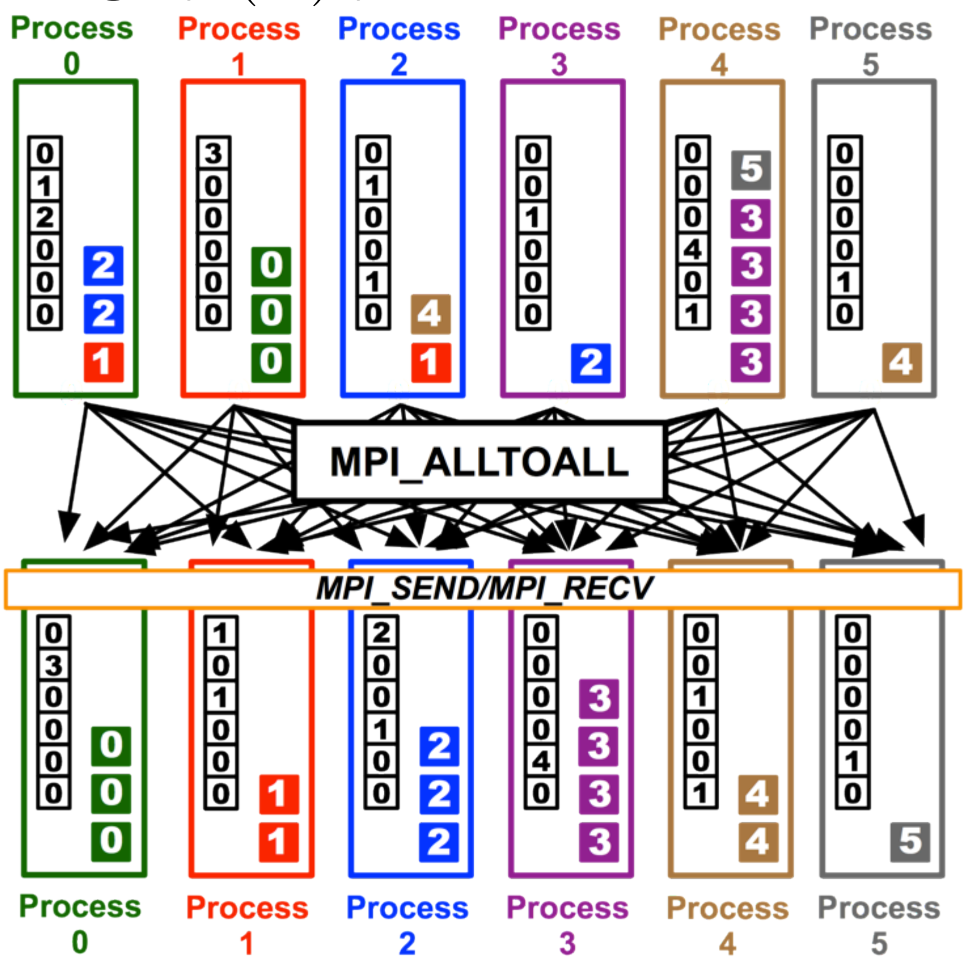
  OR:
  - Use MPI semantics:
    - Unknown sender (MPI_ANY_SOURCE)
    - Unknown message size (MPI_PROBE)
    - Reduces problem to counting the number of neighbors
    - Allow faster implementation!



Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange

Using Alltoall (PEX)

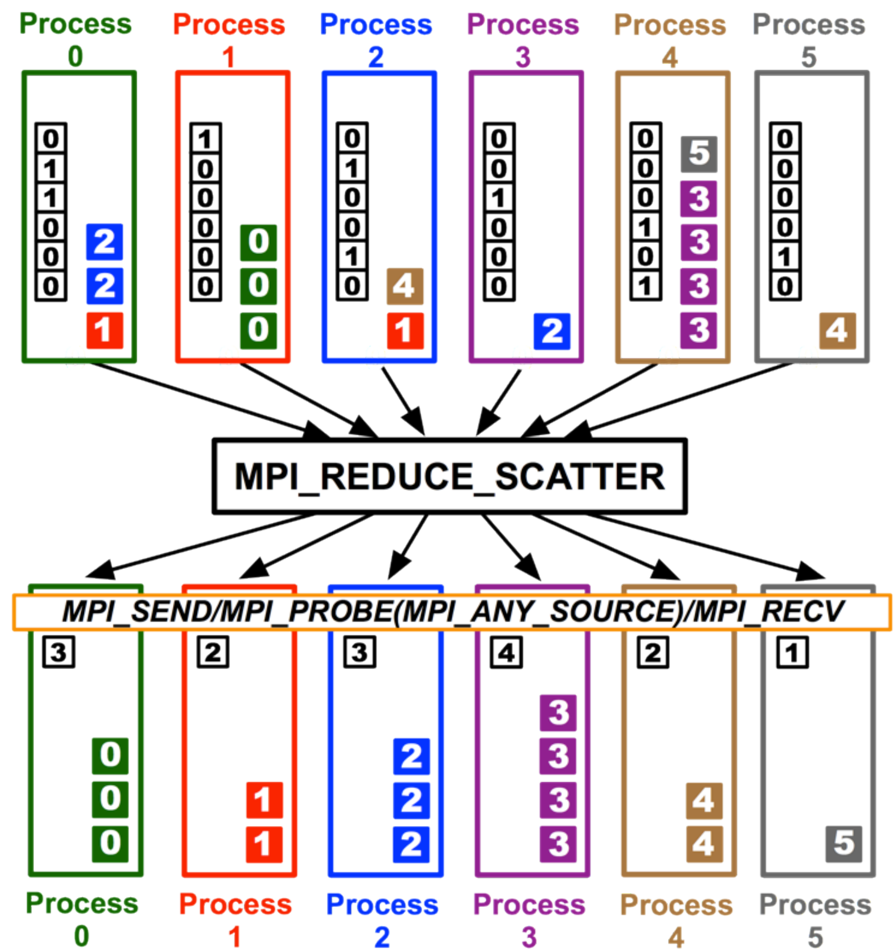
- Based on Personalized Exchange ($\Theta(P)$)
  - Processes exchange metadata (sizes) about neighborhoods with all-to-all
  - Processes post receives afterwards
  - Most intuitive but least performance and scalability!



T. Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange

Reduce_scatter (PCX)

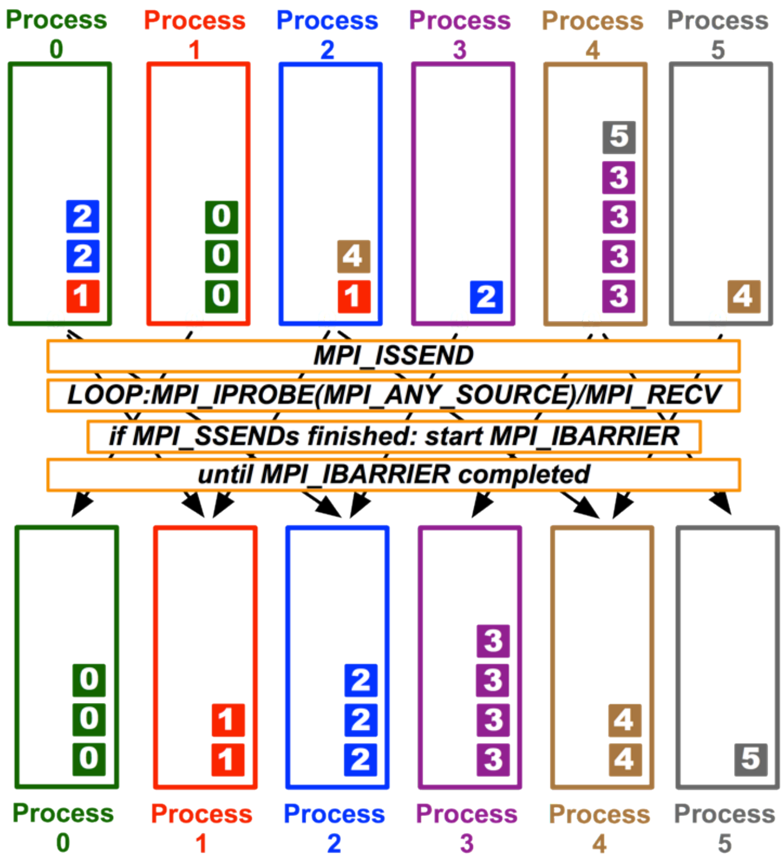
- Bases on Personalized Census ($\Theta(P)$)
  - Processes exchange metadata (counts) about neighborhoods with reduce_scatter
  - Receivers checks with wildcard MPI_IPROBE and receives messages
  - Better than PEX but non-deterministic!



T. Hoefler et al.:Scalable Communication Protocols for Dynamic Sparse Data Exchange

MPI_Ibarrier (NBX)

- Complexity - census (barrier): ($\Theta(\log(P))$)
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start coll. synch. (barrier) when p2p phase ended
    - barrier = distributed marker!
  - Better than Alltoall, reduce-scatter!

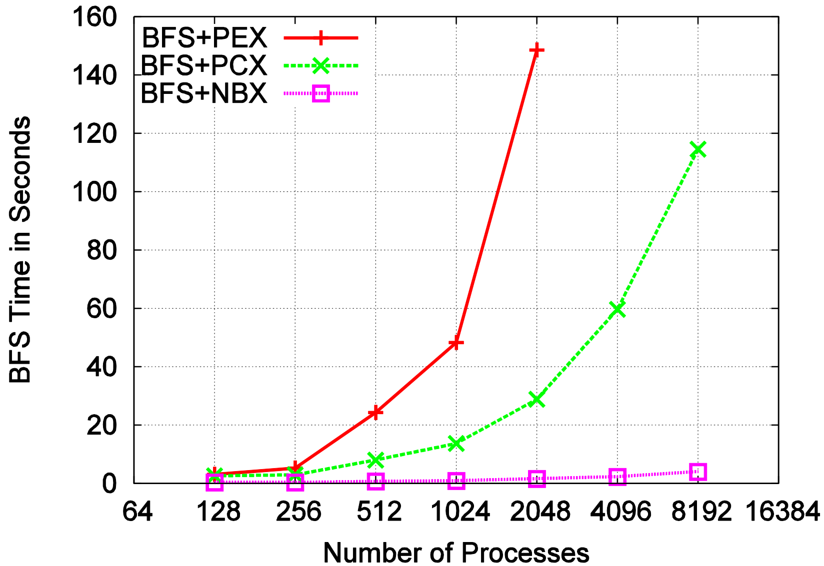


T. Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange

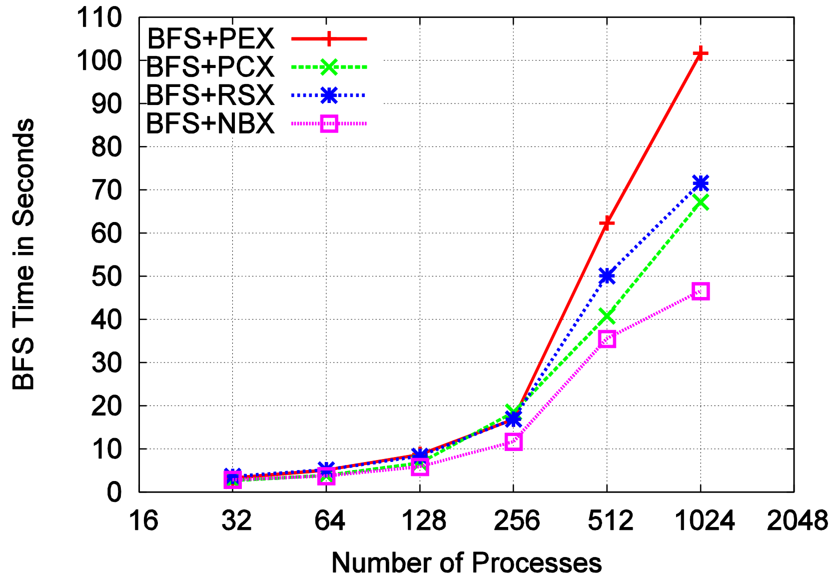
Parallel Breadth First Search

- On a clustered Erdős-Rényi graph, weak scaling
  - 6.75 million edges per node (filled 1 GiB)

BlueGene/P – with HW barrier!



Myrinet 2000 with LibNBC

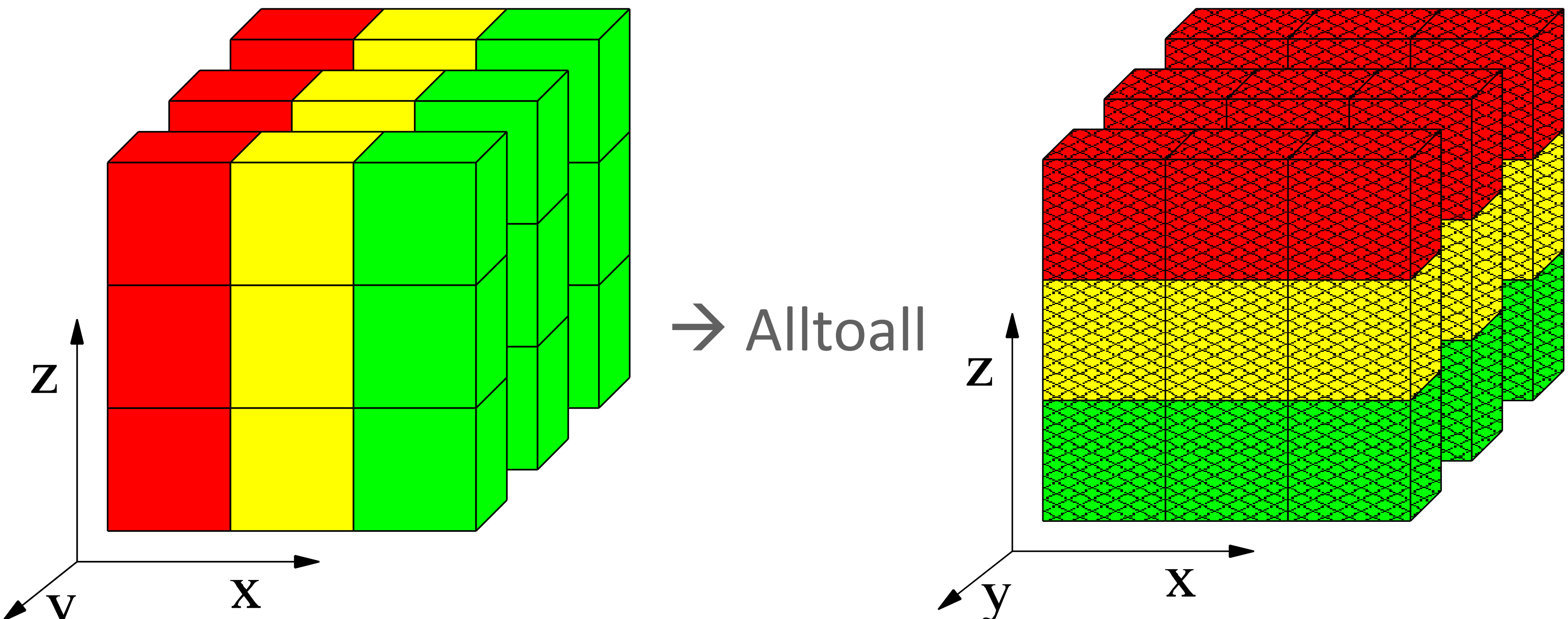


- HW barrier support is significant at large scale!

T. Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange

Parallel Fast Fourier Transform

- 1D FFTs in all three dimensions
  - Assume 1D decomposition (each process holds a set of planes)
  - Best way: call optimized 1D FFTs in parallel → alltoall



  - Red/yellow/green are the (three) different processes!

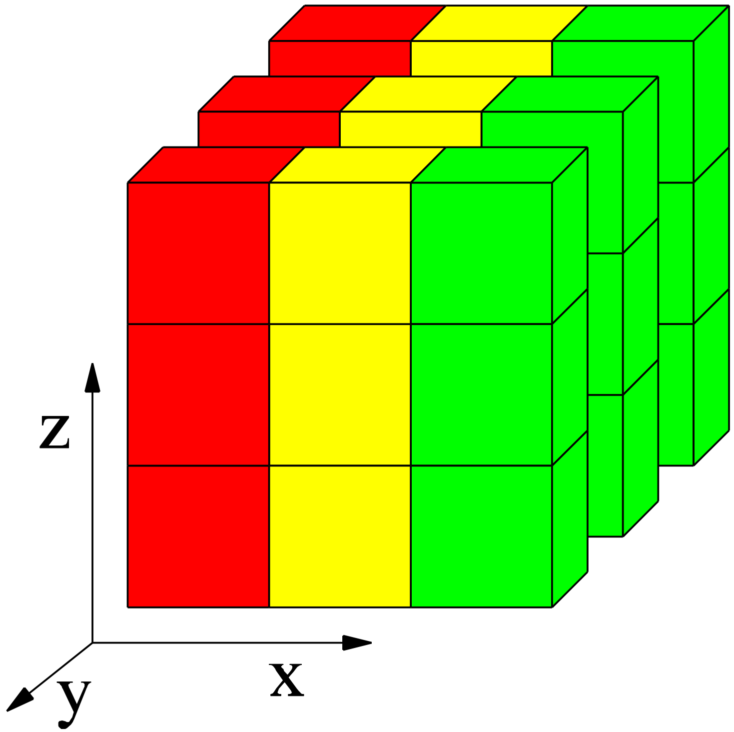
A Complex Example: FFT

```
for(int x=0; x<n/p; ++x) 1d_fft(/* x-th stencil */);

// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose

for(int y=0; y<n/p; ++y) 1d_fft(/* y-th stencil */);

// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose
```

Hoefler: Leveraging Non-blocking Collective Communication in High-performance Applications

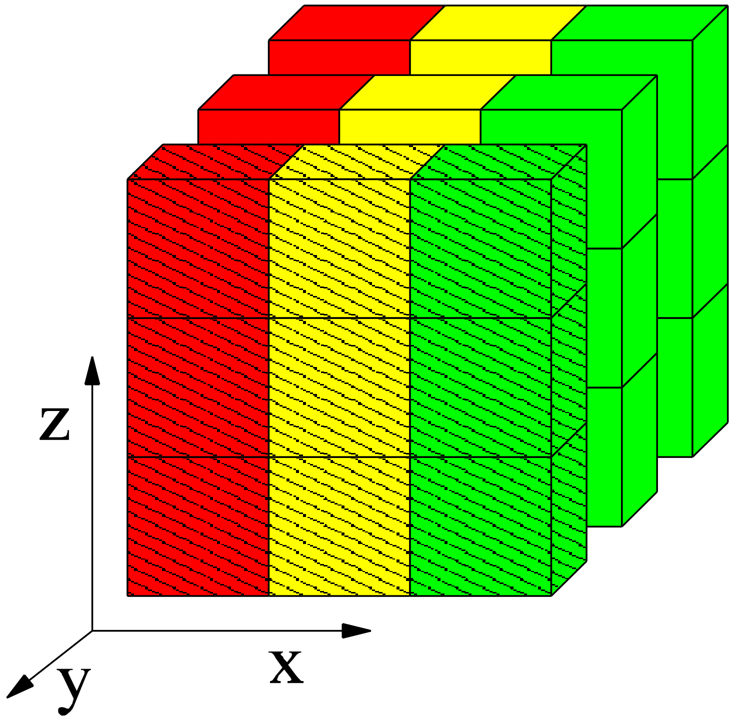
Parallel Fast Fourier Transform

- Data already transformed in y-direction



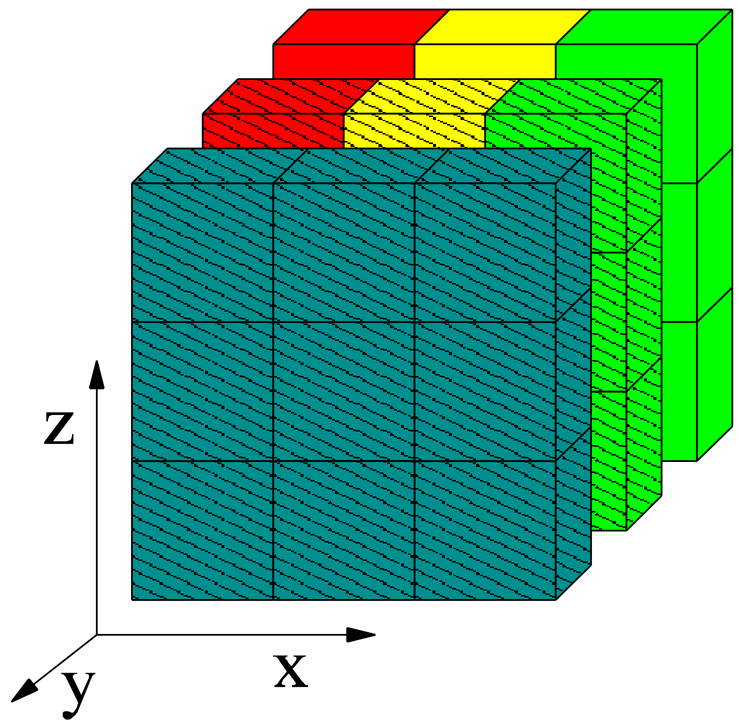
Parallel Fast Fourier Transform

- Transform first y plane in z



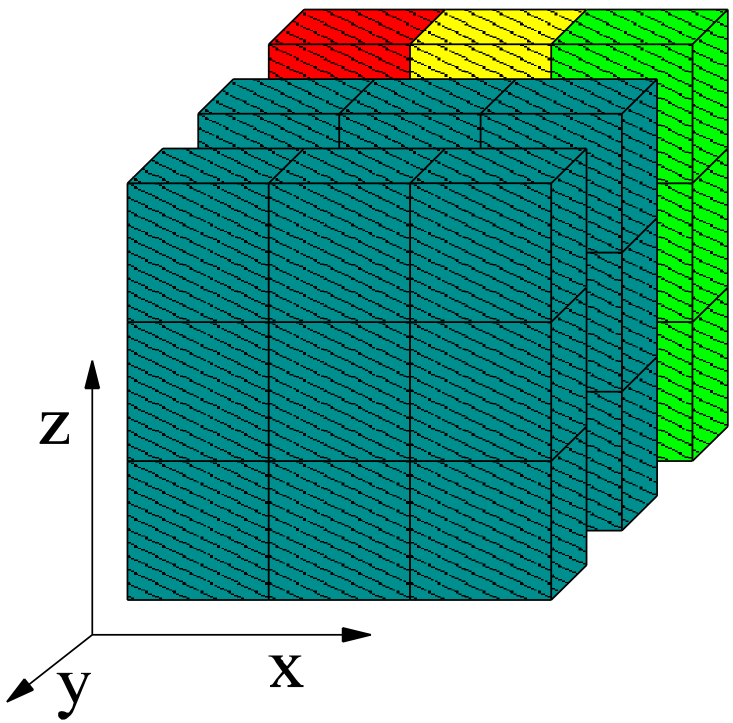
Parallel Fast Fourier Transform

- Start ialltoall and transform second plane



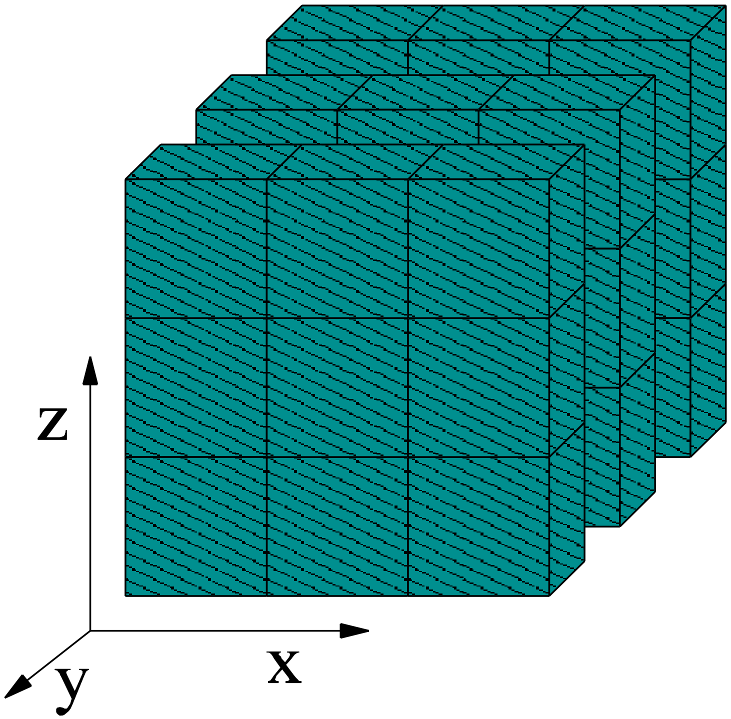
Parallel Fast Fourier Transform

- Start ialltoall (second plane) and transform third



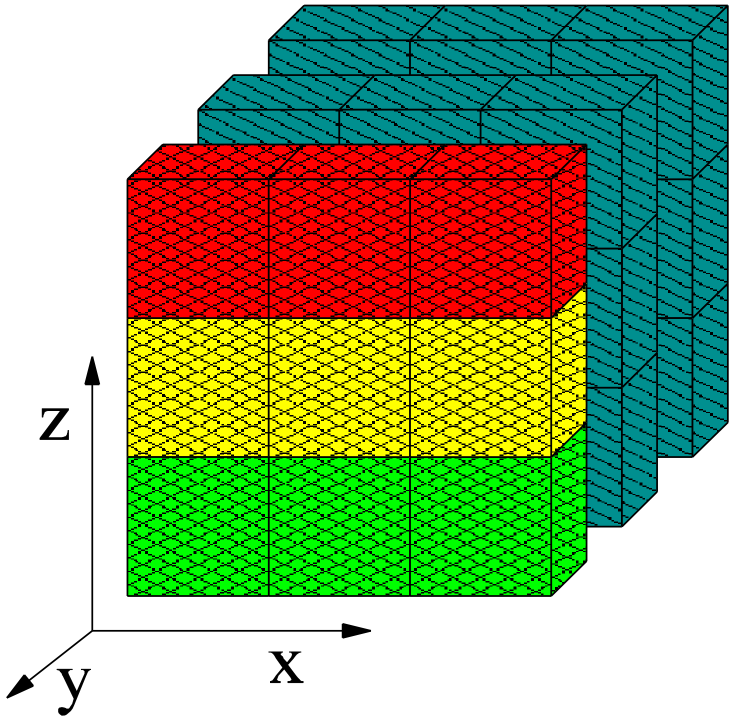
Parallel Fast Fourier Transform

- Start ialltoall of third plane and …



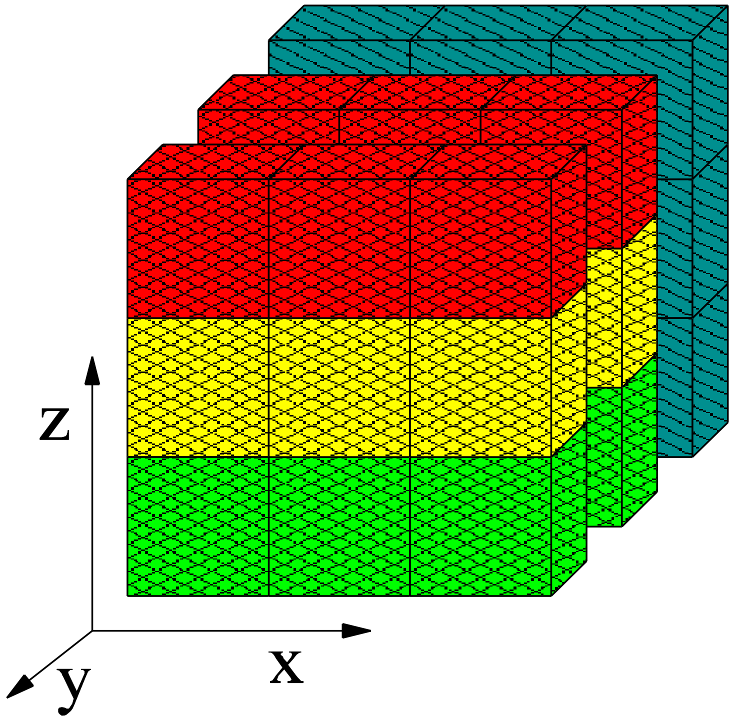
Parallel Fast Fourier Transform

- Finish ialltoall of first plane, start x transform



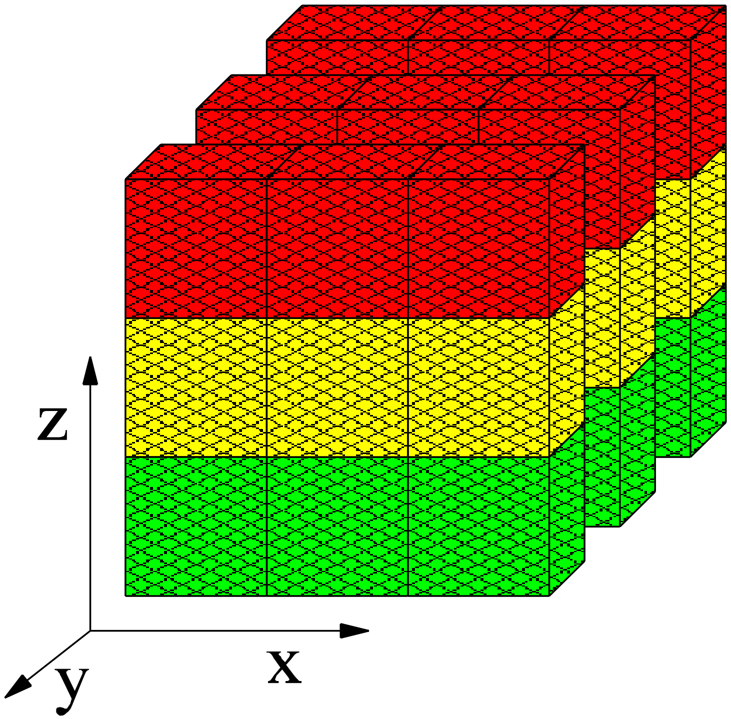
Parallel Fast Fourier Transform

- Finish second ialltoall, transform second plane



Parallel Fast Fourier Transform

- Transform last plane → done



FFT Software Pipelining

```
MPI_Request req[nb];
for(int b=0; b<nb; ++b) { // loop over blocks
  for(int x=b*n/p/nb; x<(b+1)n/p/nb; ++x) 1d_fft(/* x-th stencil*/);

  // pack b-th block of data for alltoall
  MPI_Ialltoall(&in, n/p*n/p/bs, cplx_t, &out, n/p*n/p, cplx_t, comm, &req[b]);
}
MPI_Waitall(nb, req, MPI_STATUSES_IGNORE);

// modified unpack data from alltoall and transpose
for(int y=0; y<n/p; ++y) 1d_fft(/* y-th stencil */);
// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose
```

Hoefler: Leveraging Non-blocking Collective Communication in High-performance Applications

Nonblocking Collectives Summary

- Nonblocking communication does two things:
  - Overlap and relax synchronization
- Collective communication does one thing
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- They can be composed
  - E.g., software pipelining

Topologies and Topology Mapping

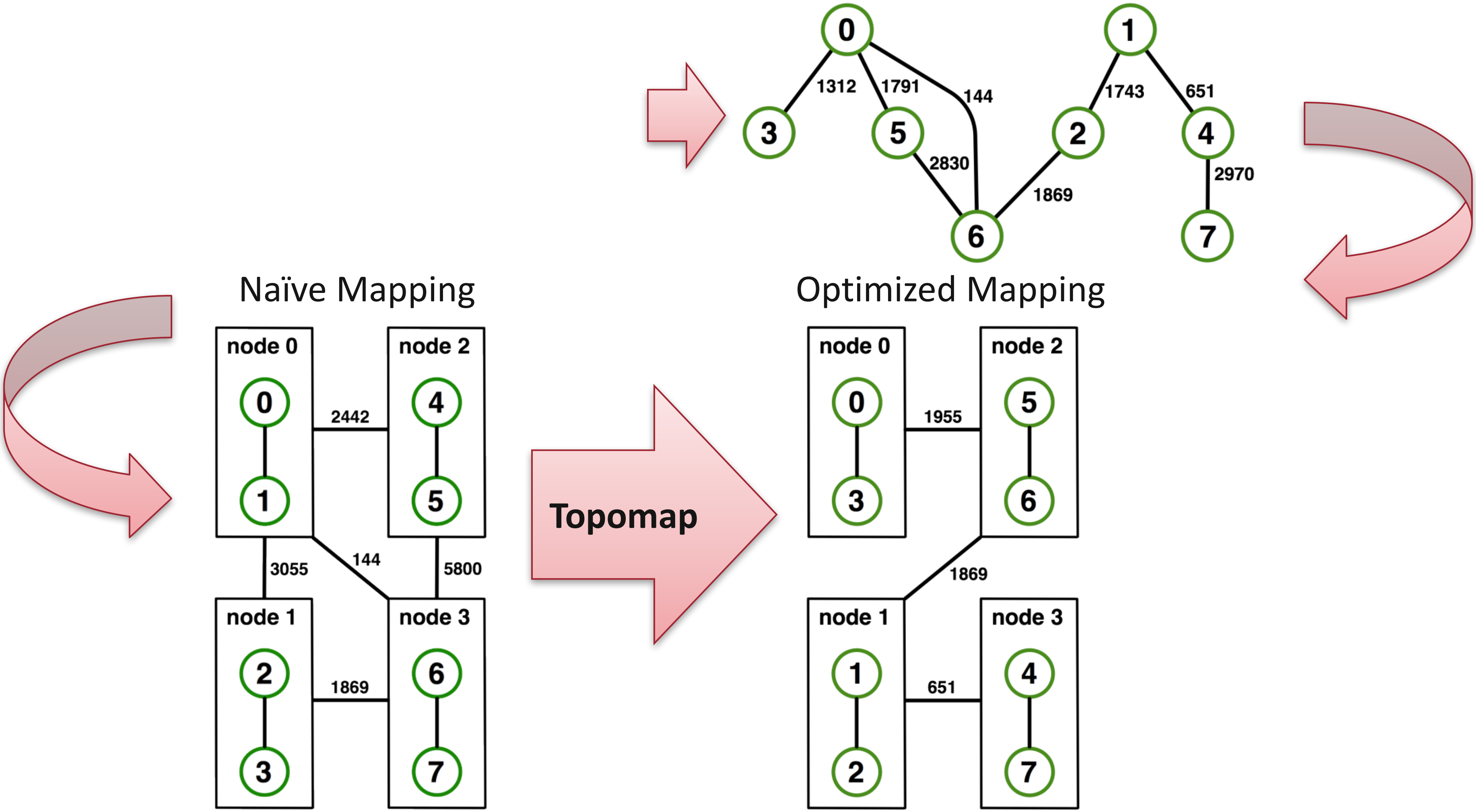
Topology Mapping and Neighborhood Collectives

- Topology mapping basics
  - Allocation mapping vs. rank reordering
  - Ad-hoc solutions vs. portability
- MPI topologies
  - Cartesian
  - Distributed graph
- Collectives on topologies – neighborhood collectives
  - Use cases

Topology Mapping Basics

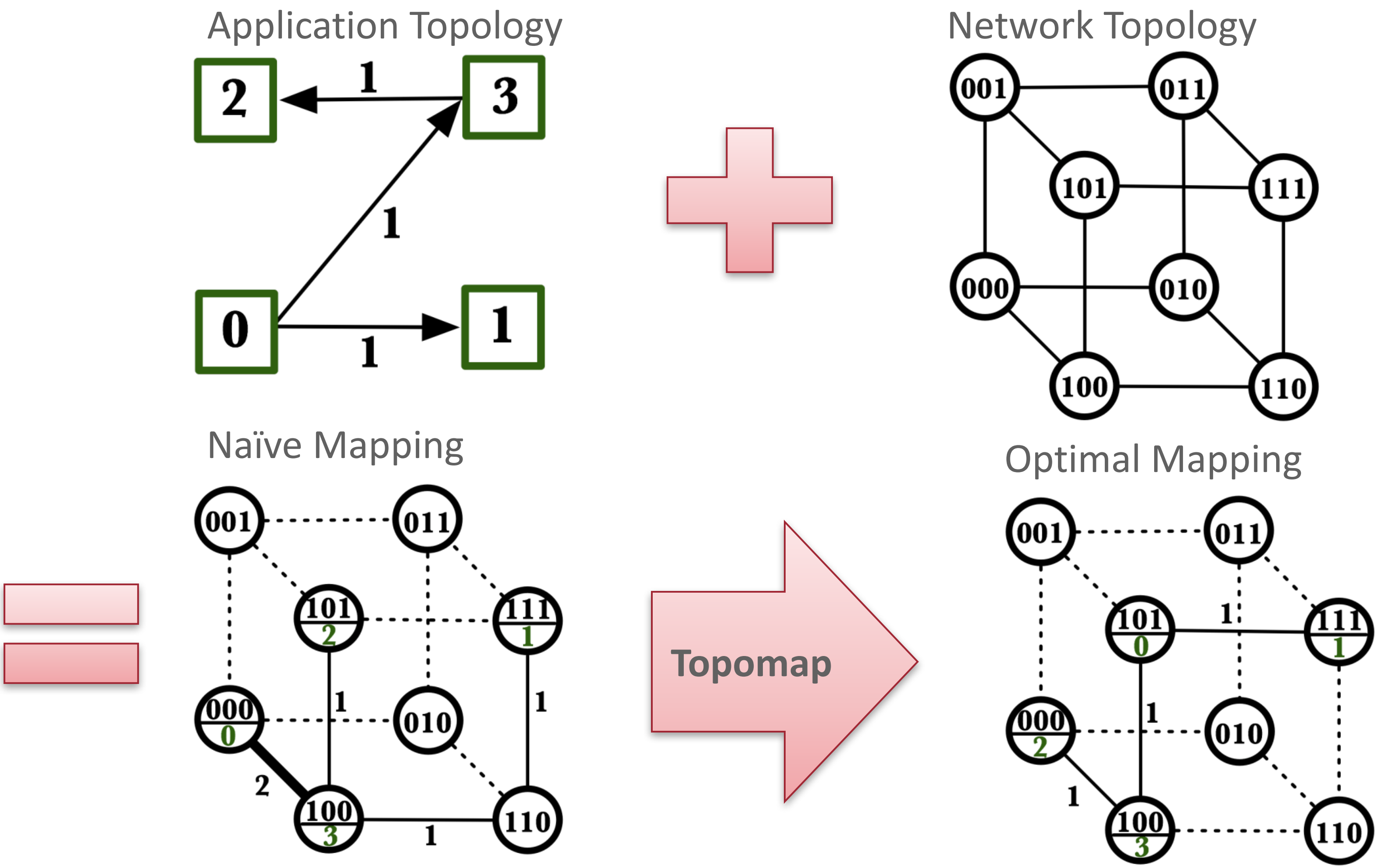
- MPI supports rank reordering
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)
- Properties
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies
    - Network topology is not exposed
  - Manual data shuffling after remapping step

Example: On-Node Reordering



Gottschling et al.: Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption

Off-Node (Network) Reordering



MPI Topology Intro

- Convenience functions (in MPI-1)
  - Create a graph and query it, nothing else
  - Useful especially for Cartesian topologies
    - Query neighbors in n-dimensional space
  - Graph topology: each rank specifies full graph ☹
- Scalable Graph topology (MPI-2.2)
  - Graph topology: each rank specifies its neighbors **or** an arbitrary subset of the graph
- Neighborhood collectives (MPI-3.0)
  - Adding communication functions defined on graph topologies (neighborhood of distance one)

MPI_Cart_create

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, const int *dims,
        const int *periods, int reorder, MPI_Comm *comm_cart)
```

- Specify ndims-dimensional topology
  - Optionally periodic in each dimension (Torus)
- Some processes may return MPI_COMM_NULL
  - Product sum of dims must be <= P
- Reorder argument allows for topology mapping
  - Each calling process may have a new rank in the created communicator
  - Data has to be remapped manually

MPI_Cart_create Example

```
int dims[3] = {5,5,5};
int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Creates logical 3D Torus of size 5 x 5 x 5
- But we're starting MPI processes with a one-dimensional argument (-p X)
  - User has to determine size of each dimension
  - Often as "square" as possible, MPI can help!

MPI_Dims_create

```
MPI_Dims_create(int nnodes, int ndims, int *dims)
```

- Create dims array for Cart_create with nnodes and ndims
  - Dimensions are as close as possible (well, in theory)
- Non-zero entries in dims will not be changed
  - nnodes must be multiple of all non-zeroes

MPI_Dims_create Example

```
int p;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Dims_create(p, 3, dims);

int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Makes life a little bit easier
  - Some problems may be better with a non-square layout though

Cartesian Query Functions

- Library support and convenience!
- MPI_Cartdim_get()
  - Gets dimensions of a Cartesian communicator
- MPI_Cart_get()
  - Gets size of dimensions
- MPI_Cart_rank()
  - Translate coordinates to rank
- MPI_Cart_coords()
  - Translate rank to coordinates

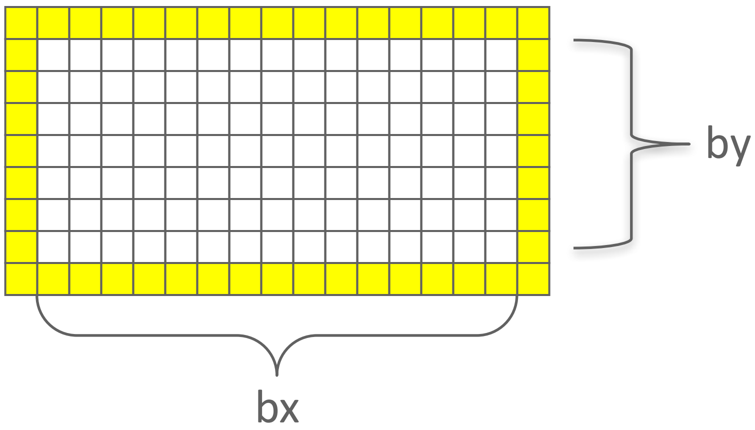
Cartesian Communication Helpers

```
MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
               int *rank_source, int *rank_dest)
```

- Shift in one dimension
  - Dimensions are numbered from 0 to ndims-1
  - Displacement indicates neighbor distance (-1, 1, …)
  - May return MPI_PROC_NULL
- Very convenient, all you need for nearest neighbor communication
  - No “over the edge” though

Code Example

- *stencil-mpi-carttopo.c*
- Adds calculation of neighbors with topology



MPI_Graph_create

```
MPI_Graph_create(MPI_Comm comm_old, int nnodes,
        const int *index, const int *edges, int reorder,
        MPI_Comm *comm_graph)
```

- Don't use!!!!!
- nnodes is the total number of nodes
- index i stores the total number of neighbors for the first i nodes (sum)
  - Acts as offset into edges array
- edges stores the edge list for all processes
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges

Distributed graph constructor

- MPI_Graph_create is discouraged
  - Not scalable
  - Not deprecated yet but hopefully soon
- New distributed interface:
  - Scalable, allows distributed graph specification
    - Either local neighbors **or** any edge in the graph
  - Specify edge weights
    - Meaning undefined but optimization opportunity for vendors!
  - Info arguments
    - Communicate assertions of semantics to the MPI library
    - E.g., semantics of edge weights

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

MPI_Dist_graph_create_adjacent

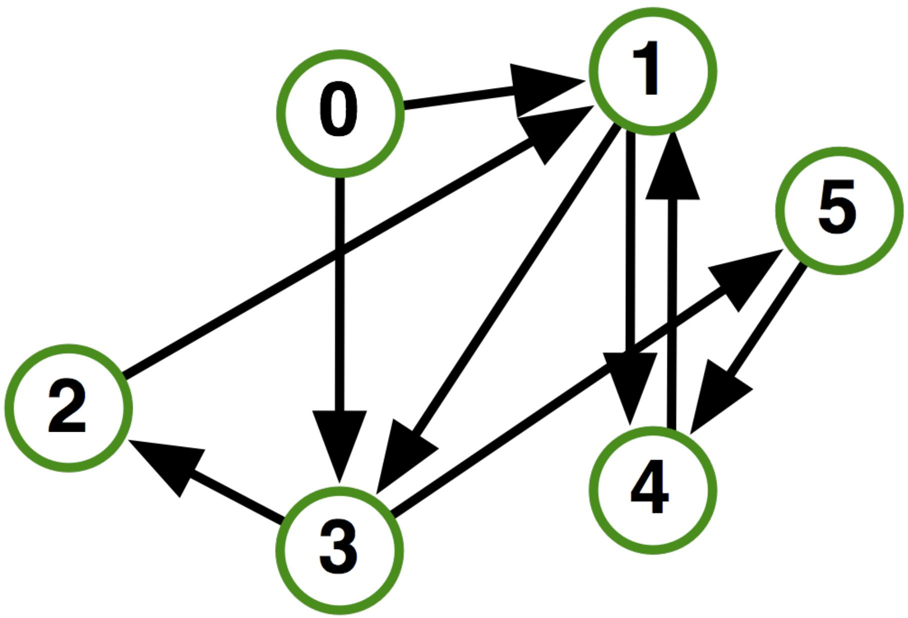
```
MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
        int indegree, const int sources[], const int sourceweights[],
        int outdegree, const int destinations[],
        const int destweights[], MPI_Info info, int reorder,
        MPI_Comm *comm_dist_graph)
```

- indegree, sources, ~weights – source proc. Spec.
- outdegree, destinations, ~weights – dest. proc. spec.
- info, reorder, comm_dist_graph – as usual
- directed graph
- Each edge is specified twice, once as out-edge (at the source) and once as in-edge (at the dest)

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

MPI_Dist_graph_create_adjacent

- Process 0:
  - Indegree: 0
  - Outdegree: 2
  - Dests: {3,1}
- Process 1:
  - Indegree: 3
  - Outdegree: 2
  - Sources: {4,0,2}
  - Dests: {3,4}
- …



Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

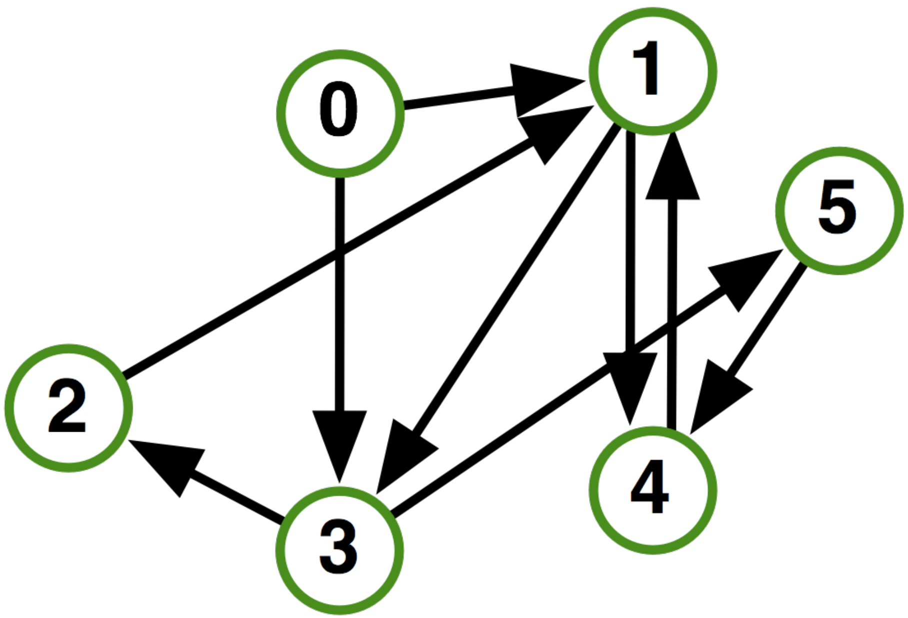
MPI_Dist_graph_create

```
MPI_Dist_graph_create(MPI_Comm comm_old, int n,
        const int sources[], const int degrees[],
        const int destinations[], const int weights[], MPI_Info info,
        int reorder, MPI_Comm *comm_dist_graph)
```

- n – number of source nodes
- sources – n source nodes
- degrees – number of edges for each source
- destinations, weights – dest. processor specification
- info, reorder – as usual
- More flexible and convenient
  - Requires global communication
  - Slightly more expensive than adjacent specification

MPI_Dist_graph_create

- Process 0:
  - N: 2
  - Sources: {0,1}
  - Degrees: {2,1} *
  - Dests: {3,1,4}
- Process 1:
  - N: 2
  - Sources: {2,3}
  - Degrees: {1,1}
  - Dests: {1,2}
- ...



* Note that in this example, process 0 specifies only one of the two outgoing edges of process 1; the second outgoing edge needs to be specified by another process

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

Distributed Graph Neighbor Queries

```
MPI_Dist_graph_neighbors_count(MPI_Comm comm,
       int *indegree,int *outdegree, int *weighted)
```

- Query the number of neighbors of **calling process**
- Returns indegree and outdegree!
- Also info if weighted

```
MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree,
       int sources[], int sourceweights[], int maxoutdegree,
       int destinations[],int destweights[])
```

- Query the neighbor list of **calling process**
- Optionally return weights

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

Further Graph Queries

```
MPI_Topo_test(MPI_Comm comm, int *status)
```

- Status is either:
  - MPI_GRAPH (ugs)
  - MPI_CART
  - MPI_DIST_GRAPH
  - MPI_UNDEFINED (no topology)
- Enables us to write libraries on top of MPI topologies!

Neighborhood Collectives

- Topologies implement no communication!
  - Just helper functions
- Collective communications only cover some patterns
  - E.g., no stencil pattern
- Several requests for “build your own collective” functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication patterns!

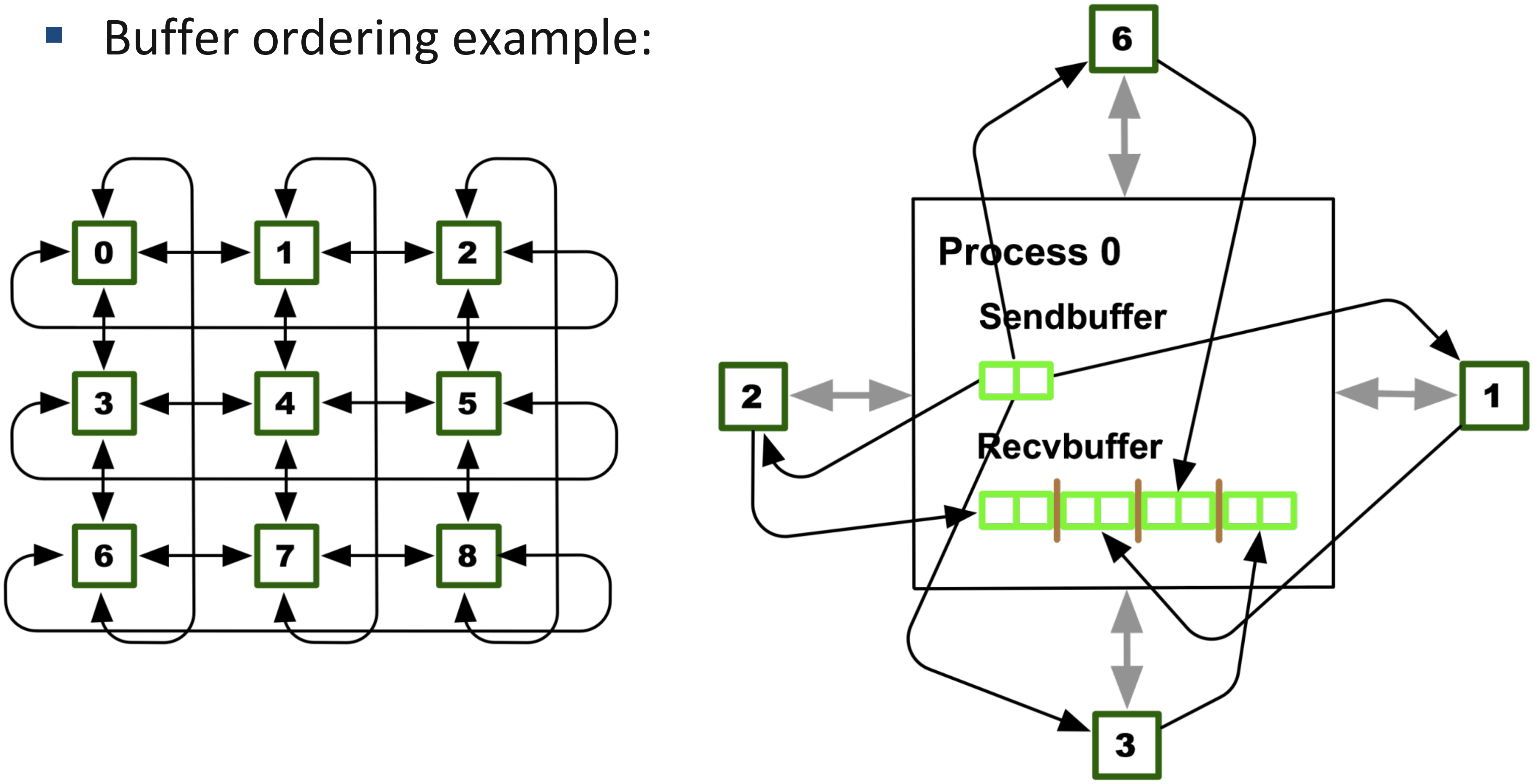
Cartesian Neighborhood Collectives

- Communicate with direct neighbors in Cartesian topology
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
  - Buffers are laid out as neighbor sequence:
    - Defined by order of dimensions, first negative, then positive
    - 2*ndims sources and destinations
    - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!

T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

Cartesian Neighborhood Collectives

- Buffer ordering example:



T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

Graph Neighborhood Collectives

- Collective Communication along arbitrary neighborhoods
  - Order is determined by order of neighbors as returned by (dist_)graph_neighbors.
  - Distributed graph is directed, may have different numbers of send/recv neighbors
  - Can express dense collective operations ☺
  - Any persistent communication pattern!

T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

MPI_Neighbor_allgather

```
MPI_Neighbor_allgather(const void* sendbuf, int sendcount,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm)
```

- Sends the same message to all neighbors
- Receives indegree distinct messages
- Similar to MPI_Gather
  - The all prefix expresses that each process is a “root” of his neighborhood
- Vector version for full flexibility

MPI_Neighbor_alltoall

```
MPI_Neighbor_alltoall(const void* sendbuf, int sendcount,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm)
```

- Sends outdegree distinct messages
- Received indegree distinct messages
- Similar to MPI_Alltoall
  - Neighborhood specifies full communication relationship
- Vector and w versions for full flexibility

Nonblocking Neighborhood Collectives

```
MPI_Ineighbor_allgather(…, MPI_Request *req);
MPI_Ineighbor_alltoall(…, MPI_Request *req);
```

- Very similar to nonblocking collectives
- Collective invocation
- Matching in-order (no tags)
  - No wild tricks with neighborhoods! In order matching per communicator!

Code Example

- *stencil_mpi_carttopo_neighcolls.c*
- Adds neighborhood collectives to the topology

Why is Neighborhood Reduce Missing?

```
MPI_Ineighbor_allreducev(…);
```

- Was originally proposed (see original paper)
- High optimization opportunities
  - Interesting tradeoffs!
  - Research topic
- Not standardized due to missing use cases
  - My team is working on an implementation
  - Offering the obvious interface

T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

Topology Summary

- Topology functions allow users to specify application communication patterns/topology
  - Convenience functions (e.g., Cartesian)
  - Storing neighborhood relations (Graph)
- Enables topology mapping (reorder=1)
  - Not widely implemented yet
  - May requires manual data re-distribution (according to new rank order)
- MPI does not expose information about the network topology (would be very complex)

Neighborhood Collectives Summary

- Neighborhood collectives add communication functions to process topologies
  - Collective optimization potential!
- Allgather
  - One item to all neighbors
- Alltoall
  - Personalized item to each neighbor
- High optimization potential (similar to collective operations)
  - Interface encourages use of topology mapping!

Section Summary

- Process topologies enable:
  - High-abstraction to specify communication pattern
  - Has to be relatively static (temporal locality)
    - Creation is expensive (collective)
  - Offers basic communication functions
- Library can optimize:
  - Communication schedule for neighborhood colls
  - Topology mapping

Please submit a tutorial evaluation

- It is useful in planning next year's tutorial program
- URL: http://bit.ly/sc17-eval
- QR Code

Recent Efforts of the MPI Forum for MPI-4 and Future MPI Standards

Introduction

- The MPI Forum continues to meet every 3 months to define future versions of the MPI Standard
- We describe some of the proposals the Forum is currently considering
- None of these topics are guaranteed to be in MPI-4
  - These are simply proposals that are being considered

MPI Working Groups

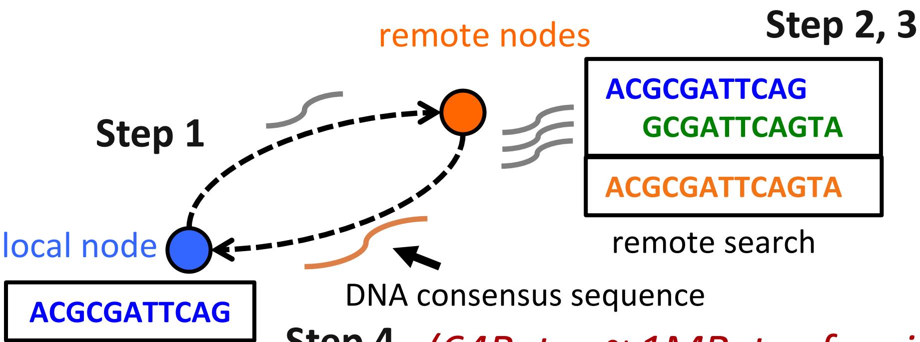
- Point-to-point communication
- Fault tolerance
- Hybrid programming
- Persistence
- Tools interfaces
- Large counts
- Others: RMA, Collectives, I/O

- http://mpi-forum.org/meetings/

Point-to-Point Working Group

Example Application: Genome Assembly

Basic edge merging algorithm



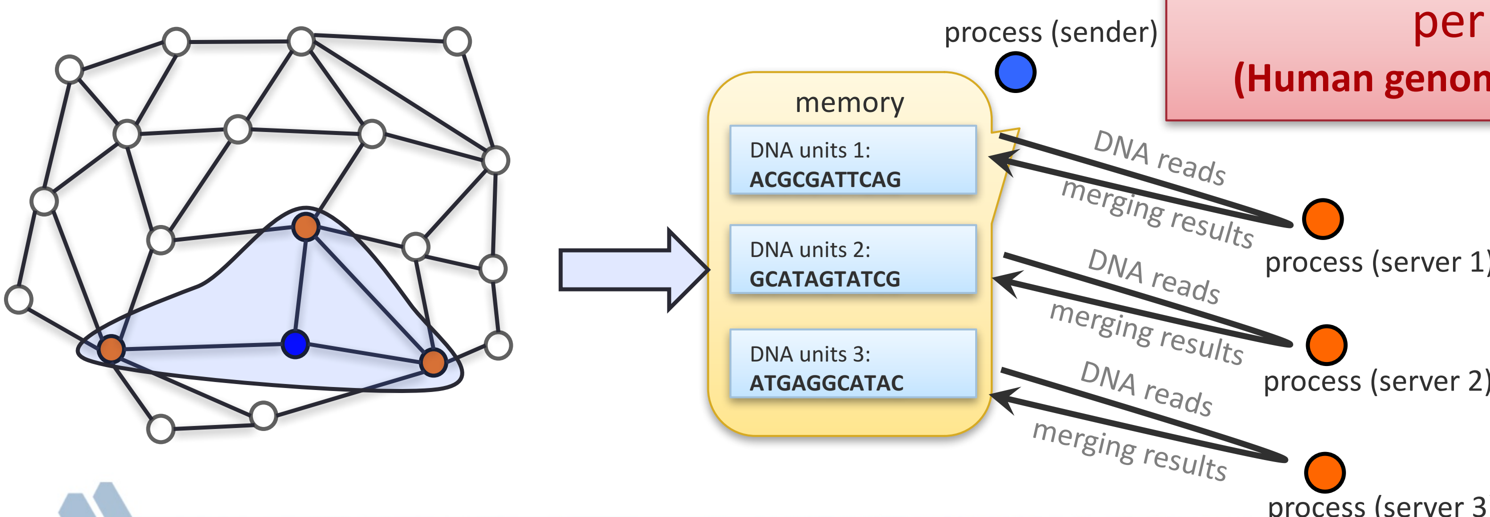
1. **Send local DNA unit to that node;**
2. **Search matching unit on that node;**
3. **Merge two units on that node;**
4. **Return merged unit.**

(64Bytes ~ 1MBytes for single message)

Large amount of outstanding data movement

10^6+ outstanding messages per process

(Human genome on Cray Edison *)



*** 64GB memory per node, 1KB memory per DNA reads, exclude runtime memory consumption.**

Proposal 1: Batched Communication Operations

- MPI-3.1 semantics
  - Each point-to-point operation creates a new request object
  - MPI library might run out of request objects after a few thousand operations
  - Application cannot issue a lot of messages to fully utilize the network
- Batched operations
  - RMA-like semantics for MPI send/recv communication
    - Application frees request as soon as the operation is issued
    - Batch completion of all operations on a communicator
      - MPI_COMM_WAITALL
  - Proportionally reduced number of requests
  - Can allow applications to consolidate multiple completions into a single request

Proposal 2: Communication Relaxation Hints

- mpi_assert_no_any_tag
  - The process will not use MPI_ANY_TAG
- mpi_assert_no_any_source
  - The process will not use MPI_ANY_SOURCE
- mpi_assert_exact_length
  - Receive buffers must be correct size for messages
- mpi_assert_overtaking_allowed
  - All messages are logically concurrent

Fault Tolerance Working Group

Improved Support for Fault Tolerance

- MPI always had support for error handlers and allows implementations to return an error code and remain alive
- MPI Forum working on additional support for MPI-4
- Current proposal handles fail-stop process failures (not silent data corruption or Byzantine failures)
  - If a communication operation fails because the other process has failed, the function returns error code MPI_ERR_PROC_FAILED
  - User can call MPI_Comm_shrink to create a new communicator that excludes failed processes
  - Collective communication can be performed on the new communicator

Proposal 1: Noncatastrophic Errors

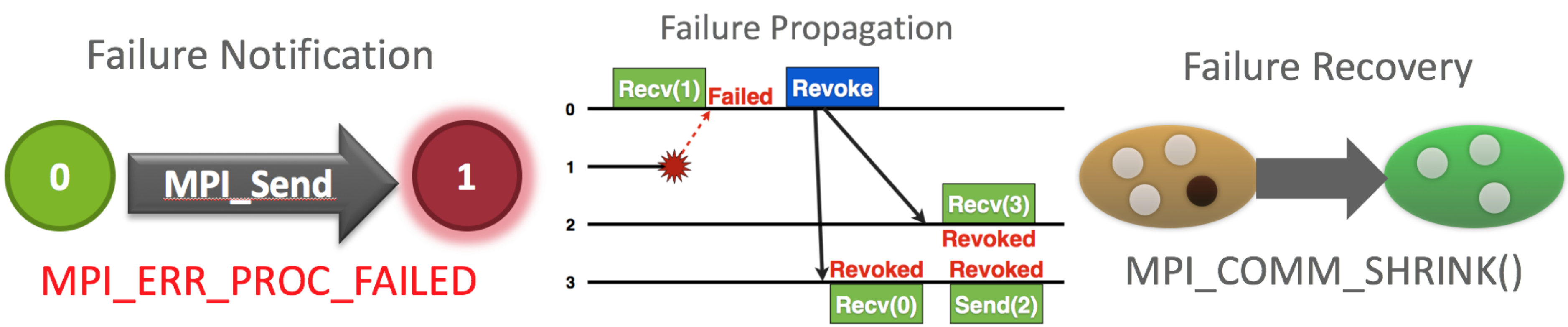
- Currently the state of MPI is undefined if any error occurs
- Even simple errors, such as incorrect arguments, can cause the state of MPI to be undefined
- Noncatastrophic errors are an opportunity for the MPI implementation to define some errors as "ignorable"
- For an error, the user can query if it is catastrophic or not
- If the error is not catastrophic, the user can simply pretend like (s)he never issued the operation and continue

Proposal 2: Error Handlers

- Cleaner semantics for error handling
- Even with MPI-3.1, errors are not always fatal
  - But semantics of error handling are cumbersome to use
  - Their specification can use more precision
- How are error handlers inherited?
- Move default error handlers from MPI_COMM_WORLD to MPI_COMM_SELF

Proposal 3: User Level Failure Mitigation

- Enable application-level recovery by providing minimal FT API to prevent deadlock and enable recovery
- Don't do recovery for the application, but let the application (or a library) do what is best.
- Currently focused on process failure (not data errors or protection)



Hybrid Programming Working Group

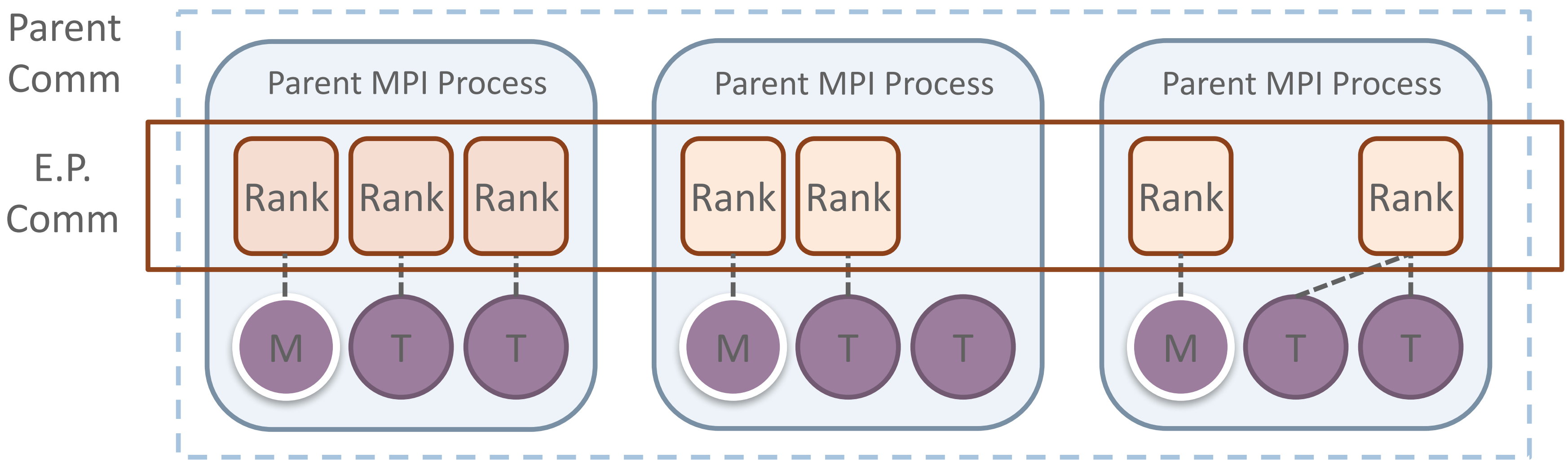
MPI-3.1 Performance/Interoperability Concerns

- Resource sharing between MPI processes
  - System resources do not scale at the same rate as processing cores
    - Memory, network endpoints, TLB entries, …
    - Sharing is necessary
  - MPI+threads gives a method for such sharing of resources
- Performance Concerns
  - MPI-3.1 provides a single view of the MPI stack to all threads
    - Requires all MPI objects (requests, communicators) to be shared between all threads
    - Not scalable to large number of threads
    - Inefficient when sharing of objects is not required by the user
  - MPI-3.1 does not allow a high-level language to interchangeably use OS processes or threads
    - No notion of addressing a single or a collection of threads
    - Needs to be emulated with tags or communicators

MPI Endpoints: Proposal for MPI-4

- Have multiple addressable communication entities within a single process
  - Instantiated in the form of multiple ranks per MPI process
- Each rank can be associated with one or more threads
- Lesser contention for communication on each “rank”
- In the extreme case, we could have one rank per thread (or some ranks might be used by a single thread)

MPI Endpoints Semantics




```
MPI_Comm_create_endpoints(MPI_Comm parent_comm, int my_num_ep,
        MPI_Info info, MPI_Comm out_comm_handles[])
```

- Creates new MPI ranks from existing ranks in parent communicator
  - Each process in parent comm. requests a number of endpoints
  - Array of output handles, one per local rank (i.e. endpoint) in endpoints communicator
  - Endpoints have MPI process semantics (e.g. progress, matching, collectives, …)
- Threads using endpoints behave like MPI processes
  - Provide per-thread communication state/resources
  - Allows implementation to provide process-like performance for threads

Persistence Working Group

Persistent Collective Operations

- An all-to-all transfer is done many times in an application
- The specific sends and receives represented never change (size, type, lengths, transfers)
- A nonblocking persistent collective operation can take the time to apply a heuristic and choose a faster way to move that data
- Fixed cost of making those decisions could be high (are amortized over all the times the function is used
- Static resource allocation can be done
- Choose fast(er) algorithm, take advantage of special cases
- Reduce queueing costs
- Special limited hardware can be allocated if available
- Choice of multiple transfer paths could also be performed

Basics

- Mirror regular nonblocking collective operations
- For each nonblocking MPI collective, add a persistent variant
- For every MPI_I<coll>, add MPI_<coll>_init
- Parameters are identical to the corresponding nonblocking variant
- All arguments "fixed" for subsequent uses
- Persistent collective operations cannot be matched with blocking or nonblocking collective calls

Init/Start

- The init function calls only perform initialization; do not start the operation
- E.g., MPI_Allreduce_init
  - Produces a persistent request (not destroyed by completion)
- Works with MPI_Start/MPI_Startall (cannot have multiple operations on the same communicator in Startall)
- Only inactive requests can be started
- MPI_Request_free can free inactive requests

Ordering of Inits and Starts

- Inits are nonblocking collective calls and must be ordered
- Persistent collective operations must be started in the same order at all processes
- Startall cannot contain multiple operations on the same communicator due to ordering ambiguity

Example

Nonblocking Collective APIs	Persistent Collective APIs

for (i=0; i<MAXITER; i++) {
  compute(bufA);
  MPI_Ibcast(bufA,...,rowcomm, &req[0]);
  compute(bufB);
  MPI_Ireduce(bufB,...,colcomm, &req[1]);
  MPI_Waitall(2, req, ...);
}
	MPI_Bcast_init(..., &req[0]);
	MPI_Reduce_init(..., &req[1]);
	for (i=0; i<MAXITER; i++) {
	  compute(bufA);
	  MPI_Start(req[0]);
	  compute(bufB);
	  MPI_Start(req[1]);
	  MPI_Waitall(2, req, ...);
	}

Tools Working Group

Active Proposals (1/2)

- New interface to replace PMPI
  - Known, longstanding problems with the current profiling interface PMPI
    - One tool at a time can use it
    - Forces tools to be monolithic (a single shared library)
    - The interception model is OS dependent
  - New interface
    - Callback design
    - Multiple tools can potentially attach
    - Maintain all old functionality
- New feature for event notification in MPI_T
  - PERUSE
  - Tool registers for interesting event and gets callback when it happens

Active Proposals (2/2)

- Debugger support - MPIR interface
  - Fixing some bugs in the original "blessed" document
    - Missing line numbers!
  - Support non-traditional MPI implementations
    - Ranks are implemented as threads
  - Support for dynamic applications
    - Commercial applications/ Ensemble applications
    - Fault tolerance
  - Handle Introspection Interface
    - See inside MPI to get details about MPI Objects
      - Communicators, File Handles, etc.

Sessions Working Group

Before MPI-3.1, this could be erroneous

```
int main(int argc, char **argv) {
    MPI_Init_thread(…, MPI_THREAD_FUNNELED, …);
    pthread_create(…, my_thread1_main, NULL);
    pthread_create(…, my_thread2_main, NULL);
    // …
}
```

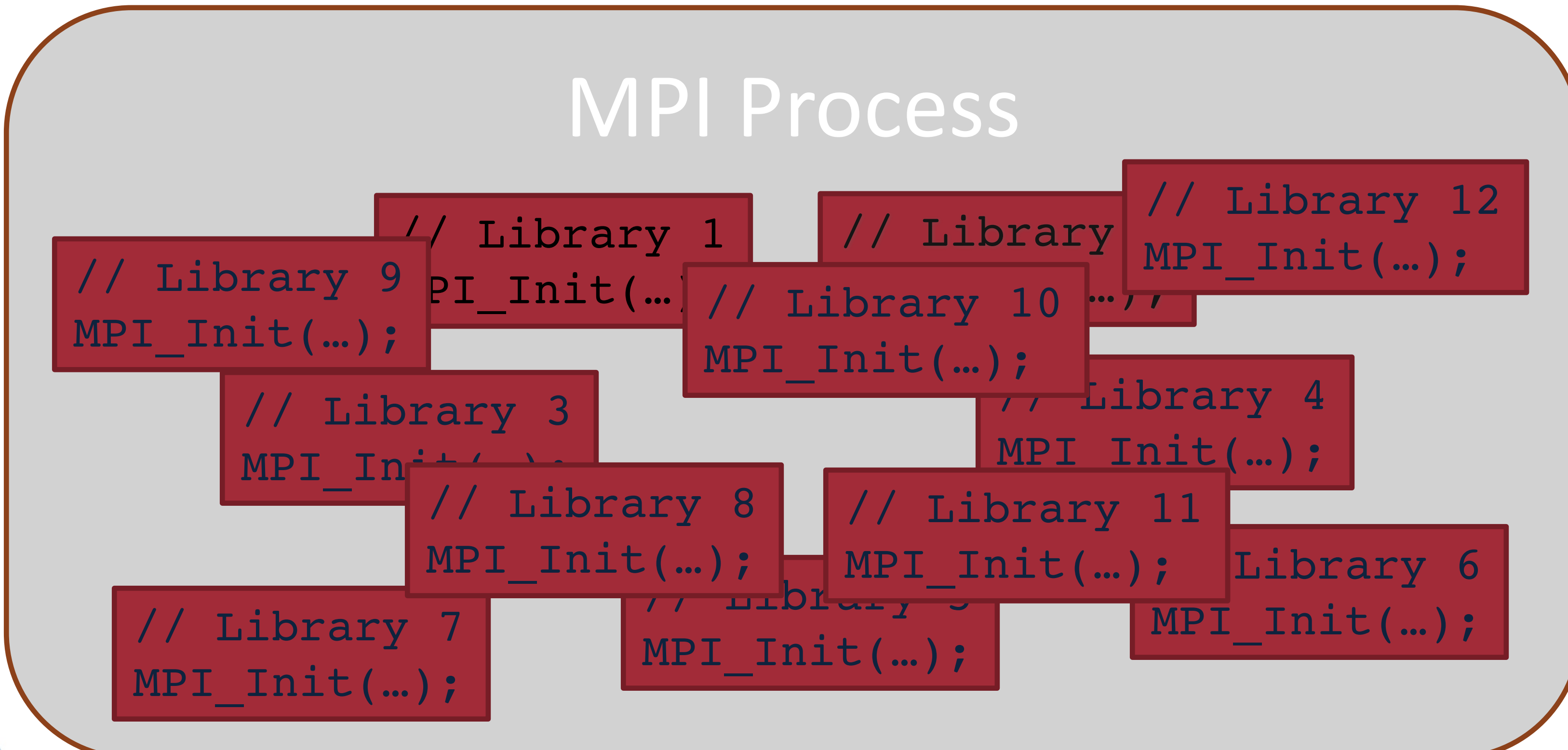
```
int my_thread1_main(void *context) {
    MPI_Initialized(&flag);
    // …
}
```

These might
run at the same time (!)

```
int my_thread2_main(void *context) {
    MPI_Initialized(&flag);
    // …
}
```

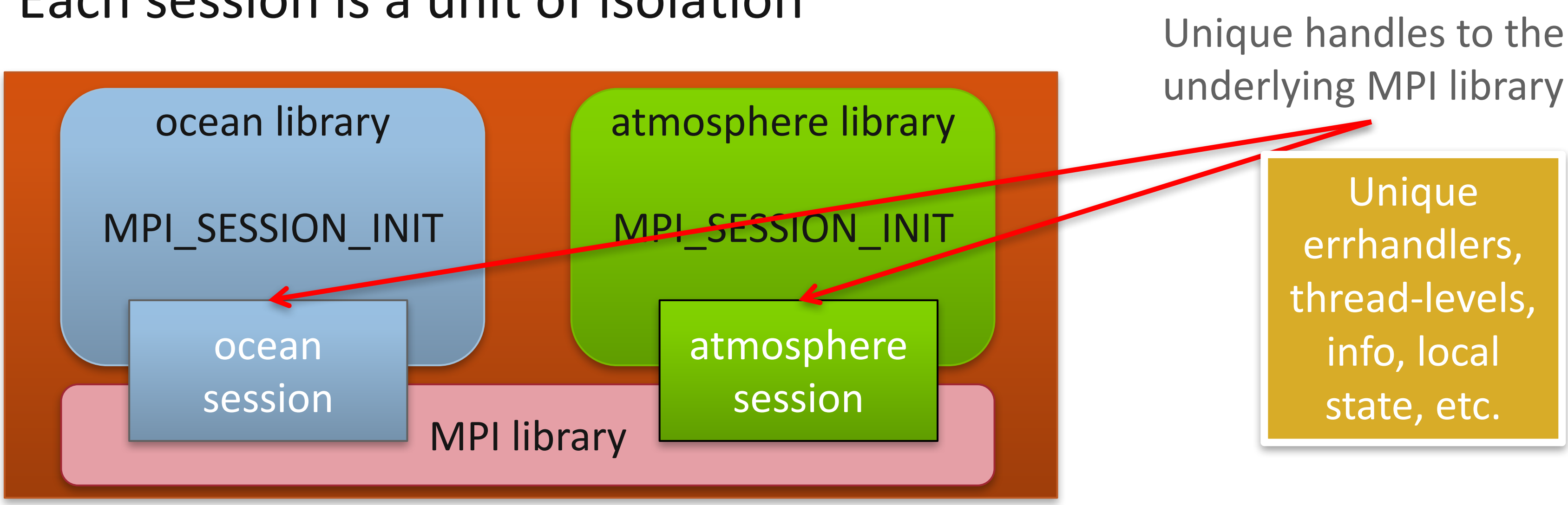
What we want

- Any thread (e.g., library) can use MPI any time it wants
- But still be able to totally clean up MPI if/when desired
- New parameters to initialize the MPI API



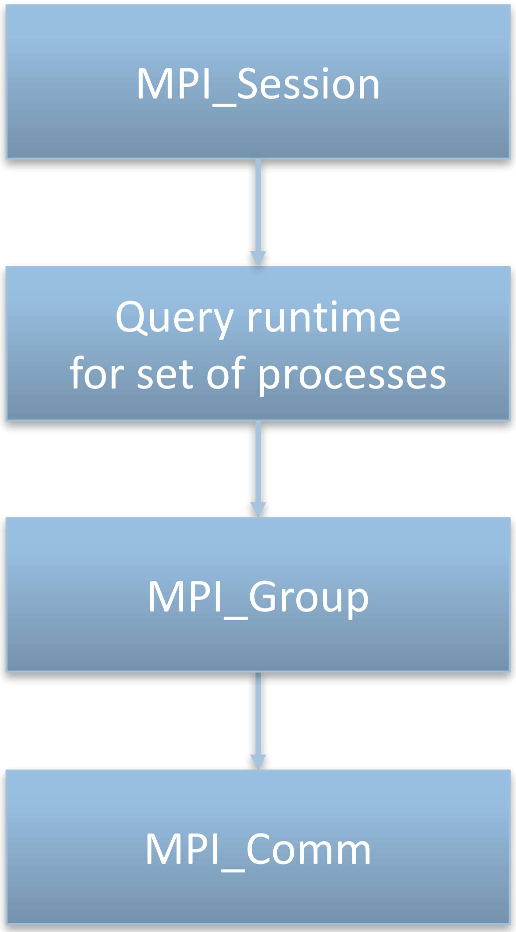
New Concept: "Session"

- A local handle to the MPI library
  - Implementation intent: lightweight / uses very few resources
  - Can also cache some local state
- Can have multiple sessions in an MPI process
  - MPI_Session_init(…, &session);
  - MPI_Session_finalize(…, &session);
- Each session is a unit of isolation



Overview

- General scheme:
  - Query the underlying run-time system
    - Get a “set” of processes
  - Determine the processes you want
    - Create an MPI_Group
  - Create a communicator with just those processes
    - Create an MPI_Comm



Static sets of processes

- Two sets are mandated to exist
    1. A set of processes effectively equivalent to the processes in MPI-3.1's MPI_COMM_WORLD
    2. A set containing only a single process
- Sets are identified by string name
    - "mpi://WORLD": refers to set #1, above
    - "mpi://SELF": refers to set #2, above
- By definition, processes will be in more than one set

Large Counts Working Group

Problem with Large Counts

- MPI_Send/Recv and other functions take “int” as the count for data
  - What happens for data larger than 2GB x datatype size?
  - You create a new large “contiguous” derived datatype and send that
  - Possible, but clumsy
- What about duplicating all MPI functions to change “int” to “MPI_Count” (which is a large, typically 64-bit, integer)
  - Doubles the number of MPI functions
  - Possible, but clumsy

New C11 Bindings

- Use C11 _Generic type to provide multiple function prototypes
  - Like C++ function overloading, but done with compile time macro replacement
- MPI_Send will have two function signatures
  - One for traditional “int” arguments
  - One for new “MPI_Count” arguments
- Fully backward compatible for existing applications
- New applications can promote their data lengths to 64-bit without changing functions everywhere

Concluding Remarks

Conclusions

- Parallelism is critical today, given that it is the only way to achieve performance improvement with modern hardware
- MPI is an industry standard model for parallel programming
  - A large number of implementations of MPI exist (both commercial and public domain)
  - Virtually every system in the world supports MPI
- Gives user explicit control on data management
- Widely used by many scientific applications with great success
- Your application can be next!

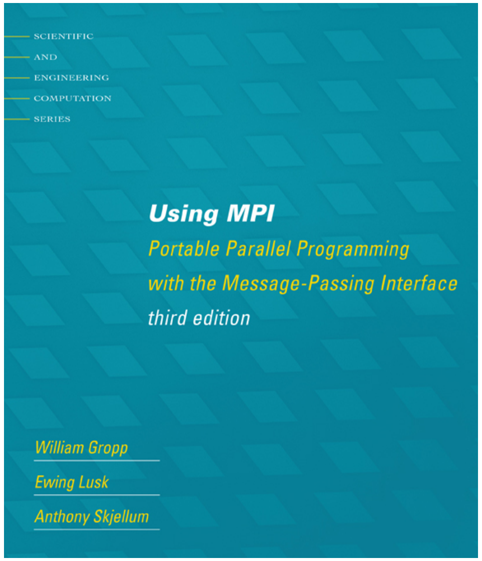
Web Pointers

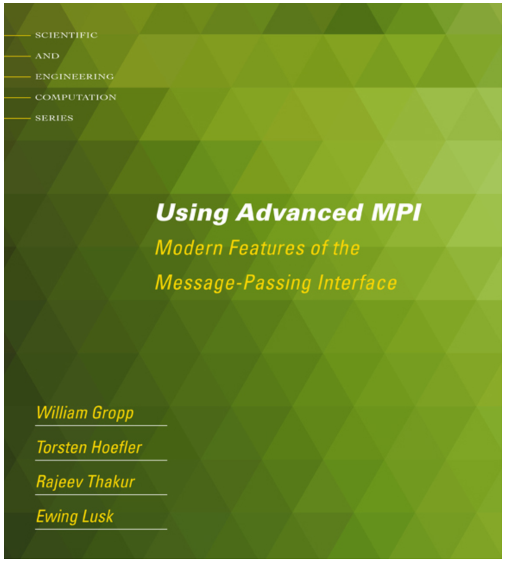
- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: https://msdn.microsoft.com/en-us/library/bb524831%28v=vs.85%29.aspx
  - Open MPI : http://www.open-mpi.org/
  - IBM MPI, Cray MPI, HP MPI, TH MPI, …
- Several MPI tutorials can be found on the web

New Tutorial Books on MPI

- For basic MPI
  - ***Using MPI, 3rd edition, 2014***, by William Gropp, Ewing Lusk, and Anthony Skjellum
  - https://mitpress.mit.edu/using-MPI-3ed
- For advanced MPI, including MPI-3
  - ***Using Advanced MPI, 2014,*** by William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk
  - https://mitpress.mit.edu/using-advanced-MPI

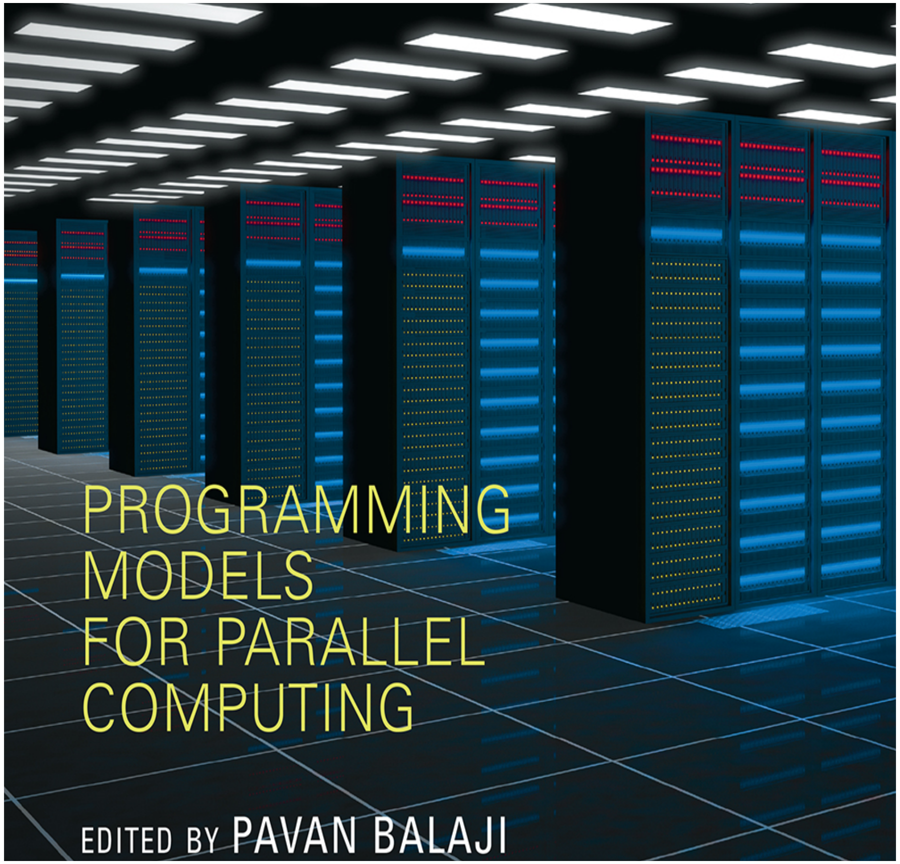




New Book on Parallel Programming Models

Edited by Pavan Balaji

- ***MPI:*** W. Gropp and R. Thakur
- ***GASNet:*** P. Hargrove
- ***OpenSHMEM:*** J. Kuehn and S. Poole
- ***UPC:*** K. Yelick and Y. Zheng
- ***Global Arrays:*** S. Krishnamoorthy, J. Daily, A. Vishnu, and B. Palmer
- ***Chapel:*** B. Chamberlain
- ***Charm++:*** L. Kale, N. Jain, and J. Lifflander
- ***ADLB:*** E. Lusk, R. Butler, and S. Pieper
- ***Scioto:*** J. Dinan
- ***SWIFT:*** T. Armstrong, J. M. Wozniak, M. Wilde, and I. Foster
- ***CnC:*** K. Knobe, M. Burke, and F. Schlimbach
- ***OpenMP:*** B. Chapman, D. Eachempati, and S. Chandrasekaran
- ***Cilk Plus:*** A. Robison and C. Leiserson
- ***Intel TBB:*** A. Kukanov
- ***CUDA:*** W. Hwu and D. Kirk
- ***OpenCL:*** T. Mattson



https://mitpress.mit.edu/models

Please submit a tutorial evaluation

- It is useful in planning next year’s tutorial program
- URL: http://bit.ly/sc17-eval
- QR Code